Microarchitecture- and Concurrency-Aware Orchestration of FaaS Workloads

Subin Hwang, Dongjoon Kim, Wonho Cho, and Won Woo Ro
Department of Electrical and Electronic Engineering, Yonsei University
Seoul, Republic of Korea
{subin.hwang, dongjoon.kim, wonho.cho, wro}@yonsei.ac.kr

***Abstract*—Serverless computing, such as Function-as-a-Service (FaaS), automates infrastructure-level provisioning, making it developer-friendly. The major overhead of FaaS, namely cold start problem, can be mitigated by leveraging container caching. FaaS platforms have been continuously optimized for renowned workload patterns such as bursty arrivals and periodic executions. However, the end-to-end (E2E) latency of concurrent serverless applications can be further reduced by exploiting microarchitectural state reuse with the above optimizations.**

In this paper, we propose a novel mechanism that optimizes the execution of functions by maximizing the reuse of microarchitectural states through the back-to-back (B2B) execution. A lightweight classifier categorizes deployed functions based on their microarchitectural characteristics. Core pools are then designated, and CPU pinning decisions are made within each pool using metadata from the classifier and runtime function queues. Based on empirical case studies, the proposed mechanism improves E2E latency up to 5% from the baseline.

***Index Terms*—Cloud computing, Computer system organization, Serverless, FaaS, Microarchitecture**

I. Introduction

Serverless computing is a developer-friendly cloud paradigm that offloads backend management responsibilities to the service provider. Function-as-a-Service (FaaS), a form of serverless computing, executes the application in a chain of functions, in contrast to the traditional monolithic application models. FaaS workloads are typically short-running and short-lived. To accommodate these properties, function executions are carried out within ephemeral and stateless sandboxes. Numerous serverless platforms adopt containerization to encapsulate and isolate functions.

Processes inherently share stateful resources, and repeatedly invoked functions often reuse function-specific data such as libraries and handler codes. However, in FaaS environments, instances of heterogeneous functions interleave at the same node, consequently thrashing shared or function-specific microarchitectural states and increasing the end-to-end (E2E) latency of a service or the series of services. Figure 1 presents the execution time and cycles per instruction (CPI) for 10 Python or Go-based functions-under-test (FUTs): authentication, CNN inference, email service, Fibonacci, AES encryption, matrix multiplication, chameleon, float operation, image processing and reservation. The experiment uses the remaining benchmark functions aside from the FUT as interleaved functions, without utilizing external resource stress tools. The inter-arrival time (IAT) sweep ranges from 100ms to 2000ms. As IAT increases, execution time increases significantly —by at least 110% and up to 150%—compared to the IAT of 100ms. This overhead is attributed to microarchitectural thrashing caused by function interleaving [1], which disrupts the efficient use of the cache hierarchy. Our proposed function orchestration design mitigates this degradation by improving microarchitectural resource reuse across function invocations.

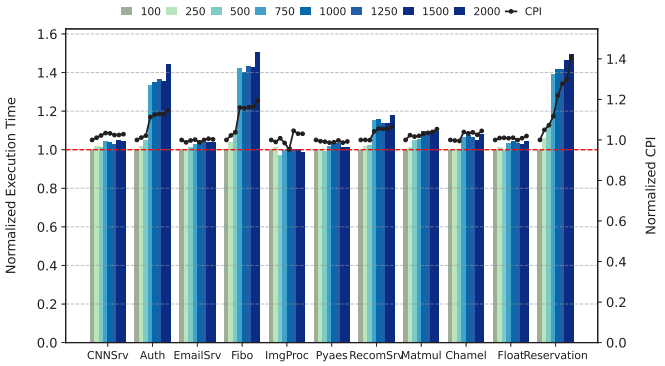

Fig. 1: Execution time and CPI under varying IAT, with interleaved functions invoked following a Poisson distribution.

II. Function Orchestration Design

Since B2B execution enables the reuse of microarchitectural state, analyzing function-level resource usage is critical. Based on this, we propose an orchestration model (Fig. 2) that classifies functions and schedules them using core pools and CPU pinning to preserve locality under high concurrency.

A. Microarchitectural Function Analysis and Classification

To characterize the microarchitectural behavior of each function, L1/L2 data and instruction misses, as well as L3 cache misses, are collected across the IAT sweep using dummy invocations and stored as metadata. Overall, cache misses increase by at least 10% and up to 300% as IAT increases. The absolute values and the increased cache miss percentage of cache misses vary between functions, enabling function classification based on both the execution time sensitivity to IAT and the cache miss behavior. For example, CNNsrv and ImgProc exhibit considerably high cache miss compared to other functions and, consequently, show a small execution time variance with respect to IAT, as depicted in Figure 1. Therefore, they fall under the same category, expected to

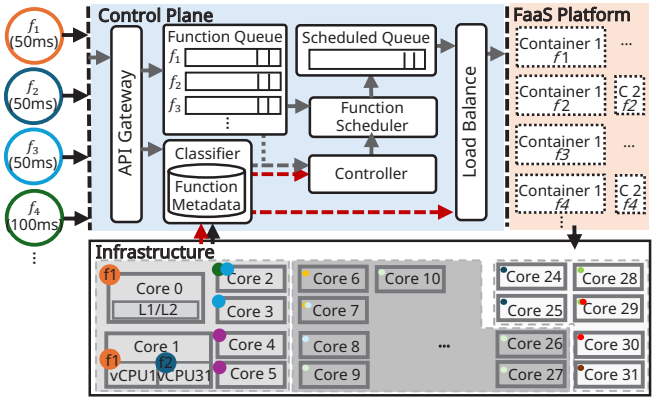

Fig. 2: Overall illustration of the mechanism and its interaction with the FaaS platform and underlying infrastructure.

benefit less by B2B execution. In contrast, Auth and Fibo demonstrate the opposite behavior, with greater sensitivity to interleaving. These observations support the classification of functions based on their microarchitectural characteristics.

B. Controller and Scheduler: Core Pool and CPU Pinning

The classification information and real-time function traffic are jointly considered to determine core pools and apply CPU pinning. Functions that leverage the most from B2B execution are pinned to the dedicated CPUs to maximize microarchitectural state reuse. CPU pinning is applied in container granularity to preserve locality of the stateful resources and reduce context-switching overhead, even in horizontally scaled-out environments. On the other hand, functions with low B2B execution sensitivity are scheduled without CPU pinning to maintain scheduling flexibility.

Based on core pool assignments and CPU pinning decisions, the scheduler reorders function invocations, prioritizing pinned functions according to QoS requirements. Pinned functions are necessarily executed with the least queuing delay, so they are dispatched through a dedicated scheduling path. In particular, high-priority pinned functions are routed to a separate queue, while low-priority ones may tolerate a certain delay.

III. Evaluation

A. Methodology and Workloads

All experiments are conducted on a dual-socket AMD EPYC 7551 server, comprising two 32-core processors running at 2GHz. Each core has a private 32 KB instruction and 64 KB data L1 cache and 512 KB L2 cache, while the system shares a 64 MB L3 cache. With simultaneous multithreading (SMT) enabled, the server runs Ubuntu 24.04.2 LTS. The FaaS environment is deployed on Kubernetes (K8S) [2] using the OpenFaaS framework, and hardware-level metrics are collected using the *perf*, *taskset* and Linux *cgroup* utilities.

Functions under the scope are selected from FunctionBench [3] and vSwarm [4], two widely used serverless benchmarking suites. To better reflect realistic scenarios —particularly in terms of function concurrency and periodicity— we adapt the Azure Functions trace [5] for our experimental setup.

B. Experimental Results

For the experimental evaluation, the proposed schemes were applied incrementally to observe the impact of each. Two baseline configurations without core pooling or CPU pinning were used, employing 5 and 10 cores, respectively. Figure 3 presents the normalized E2E latencies relative to the 5-core baseline. Core pooling alone achieved average E2E speedups of 3.1% (5-core) and 3.5% (10-core), while adding CPU pinning further improved the speedups to 2.5% and 4.9%, respectively. The 5-core environment was more sensitive to scheduling policies: only one trace (Trace 5) showed improvement when applying the additional pinning scheme. In contrast, the 10-core environment consistently achieved speedups across all tested traces, demonstrating greater robustness to the orchestration scheme.

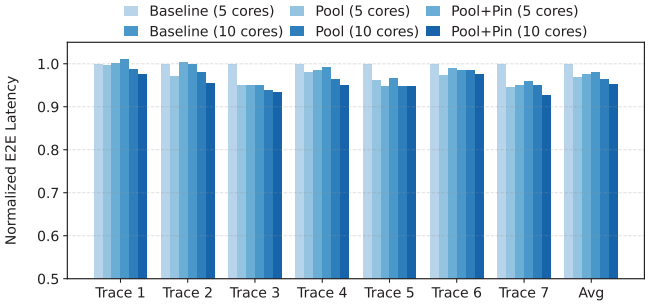

Fig. 3: E2E latency for different function traces for 10 and 5 cores. Core pooling and pinning are added to the baseline.

IV. Conclusion

FaaS workloads exhibit burstiness and repeated concurrent invocations of identical functions, creating opportunities to exploit hot microarchitectural states. By assigning core pools and applying CPU pinning per function, back-to-back executions are maximized, reducing end-to-end latency. Evaluation shows an average of 5% speedup for short-duration traces.

Acknowledgment

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2024-00402898, Simulation-based Highspeed/High-Accuracy Data Center Workload/System Analysis Platform)

References

[1] Kubernetes. 2025. kubernetes.io. Accessed: Sep 16, 2025.
[2] D. Schall, A. Margaritov, D. Ustiugov, A. Sandberg, and B. Grot, "Lukewarm Serverless Functions: Characterization and Optimization," in Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22), 2022.
[3] J. Kim and K. Lee, "FunctionBench: A Suite of Workloads for Serverless Cloud Function Service," in Proceedings of the IEEE 12th International Conference on Cloud Computing (CLOUD '19), 2019.

[4] The vHive Ecosystem, “vSwarm-Serverless Benchmarking Suite,” July 2025. [Online]. Available: https://github.com/vhive-serverless/ vSwarm
[5] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, Ricardo Bianchini. ”Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider”, in Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 20). USENIX Association, Boston, MA, July 2020.