CROSS-LAYER RESOURCE MANAGEMENT FOR QOS IN SERVERLESS PLATFORMS

ABHISEK PANDA



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY DELHI

JULY 2026

© **Indian Institute of Technology Delhi (IITD), New Delhi, 2026**

CROSS-LAYER RESOURCE MANAGEMENT FOR QOS IN SERVERLESS PLATFORMS

by

ABHISEK PANDA

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Submitted

in fulfillment of the requirements of the degree of **Doctor of Philosophy**

to the



Indian Institute of Technology Delhi

JULY 2026

For my parents,

Shri Santosh Kumar Panda and Shrimati Prabeena Kumari Panda.

Certificate

This is to certify that the thesis titled **CROSS-LAYER RESOURCE MANAGEMENT FOR QOS IN SERVERLESS PLATFORMS** being submitted by **Mr. Abhisek Panda** for the award of **Doctor of Philosophy** in **Computer Science** is a record of bona fide work carried out by him under my guidance and supervision at the **Department of Computer Science and Engineering, Indian Institute of Technology Delhi**. The work presented in this thesis has not been submitted elsewhere, either in part or full, for the award of any other degree or diploma.

Smruti Ranjan Sarangi

Professor

Department of Computer Science and Engineering

Indian Institute of Technology Delhi

New Delhi- 110016

Acknowledgements

Improving yourself by just 1% each day may seem small, but over the course of a year, it compounds into a remarkable transformation – making you nearly 37 times better than where you started.

James Clear

My Ph.D. journey has been a mix of successes and challenges. While the successes kept me motivated, it was the challenges that shaped me the most and taught me invaluable lessons. This experience has strengthened my perseverance and helped me understand that rejections and setbacks are an integral part of life. These are the entities that push us beyond our comfort zones to perform better each time. As I conclude my doctoral tenure, I feel that I have not only gained strong research skills but have also grown as a person.

I want to start by expressing my sincere gratitude to my Ph.D. supervisor, **Dr. Smruti R. Sarangi**, for his guidance and mentorship, which have played an important role in my growth as a researcher. I am particularly thankful for encouraging me to articulate problems and solutions in a clear, concise, and often mathematical manner, enabling the core ideas to be communicated effectively. This guidance has shaped all of my work and greatly enhanced the way I present my thoughts. I have also greatly valued our discussions on life, ethics, and professionalism.

I am equally grateful to my Student Research Committee members, **Prof. Sorav Bansal**, **Prof. Abhilash Jindal**, and **Prof. Arpan Chattopadhyay**, for their insightful feedback and continuous support throughout my research. I would also like to thank the administrative staff in the department – Mr. Hemant Prasad, Ms. Manju Rani, Ms. Rekha Rathee, Ms. Vandana Ahluwalia, and Ms. Aditi Jain – for their assistance.

During the course of my Ph.D., I found many great friends: Dr. Sandeep Kumar, Dr. Sahil Manchanda, Dr. Sidharth Ranjan, Dr. Akanksha Dixit, Shruti Pandey, Rahul Kanyal, Tanvi Singla, Sonali Rout, Subhankar Sarkar, Raj Kamal, Siddharth Shrivastava, Soham Nag, Rohith Vishnumolakala, Amit Prasad, Nikhil Jangamreddy, Dr. Priyanka Singla, Dr. Shubhankar Suman, Dr. Dishant Goel, and Dr. Nivedita Shrivastava. I am especially thankful to **Dr. Sandeep**

Kumar for his collaboration on several projects and for countless insightful discussions. His approach to problem-solving, resilience, and clarity in research methodology have a lasting influence on the way I think about and conduct research. I would like to thank Jayant Mishra, Kshitij Kotasthane, and Manas Mishra for standing by me throughout this journey and for all the hostel-room parties that acted as stress busters. I am thankful to Megha Panigrahi for the constant encouragement and timely push to complete and submit my thesis.

Finally, I wish to express my deepest gratitude to my parents, Mr. Santosh Kumar Panda and Mrs. Prabeena Kumari Panda, for their unwavering love and support throughout my life — none of this would have been possible without them. I am also grateful to my younger brother, Mr. Abhilas Panda, for taking care of family responsibilities during this time. I am also sincerely thankful to my grandparents, uncles, aunts, and cousins for their constant encouragement. I am especially grateful to my grandfather, Mr. Biswanath Panda, for his boundless affection, faith in me, and belief that I could achieve anything. I only wish he were here to share this moment.

Abhisek Panda

Abstract

Serverless computing has quickly become a mainstream paradigm for cloud applications, primarily due to its ability to liberate developers from infrastructure management while providing fine-grained billing and autoscaling. Unlike traditional cloud models, serverless platforms execute event-driven functions in lightweight sandboxes, allowing users to pay only for the duration of actual execution. This simplicity, coupled with elasticity, has driven its widespread adoption by major cloud providers such as Amazon, Microsoft, Google, and IBM, powering domains like IoT, machine learning, data analytics, and large-scale portals. However, this shift also exposes a new set of challenges. Serverless workloads are extremely diverse: some demand strict latency guarantees (latency-sensitive class), while others prioritize throughput (best-effort class). Meeting these QoS requirements on shared multi-tenant infrastructures is difficult, as the serverless execution pipeline introduces bottlenecks at multiple stages that interfere with predictable performance. This thesis examines the function execution pipeline step by step, identifying the major sources of overhead and proposing mechanisms to mitigate them.

When a function request arrives, the platform must first ensure that its disk image is available on the executing node. Owing to the significant variability in request inter-arrival times, images are less likely to be found locally, forcing platforms to frequently retrieve them from remote registries. This image retrieval step is the initial bottleneck, particularly for latency-sensitive (LS) applications that demand predictable performance. To resolve this issue, we propose FaaSImage, a framework that reengineers function images to minimize image retrieval overheads. FaaSImage integrates three mechanisms: layer subsetting, which downloads only the files required for startup; on-demand downloading, which lazily retrieves additional files during execution; and layer fusion, which maximizes overlap across functions to avoid redundant transfers. Collectively, these mechanisms accelerate image pulls, thereby improving cold start performance in serverless platforms.

Once images are available, the platform must perform sandbox initialization, which further threatens QoS guarantees. For LS applications having sub-second execution latencies, the la-

tency of sandbox creation can dominate response time, thereby leading to QoS violations. To mitigate this problem, prior work proposed snapshot and restore-based mechanisms, where a function snapshot captures the memory and CPU state of a sandbox, enabling faster warm starts by avoiding repeated initialization. However, the size of a snapshot is proportional to the sandbox's memory size, and storing many of them introduces larger disk overheads, especially when hundreds of functions are co-located. To address this, we introduce SnapStore, a memory region-aware snapshot deduplication framework that increases the snapshot density on a node. SnapStore exploits the structure of a program's memory map to identify duplicates efficiently and reduce hashing costs, thereby improving the deduplication latency. Moreover, it accelerates snapshot restoration by sequentializing disk accesses and leveraging RAM-based caches. Therefore, SnapStore reduces storage costs, accelerates restoration, and ensures that sandbox creation no longer dominates end-to-end performance.

With image retrieval and sandbox initialization addressed, the next challenge is resource contention during execution. Serverless platforms execute latency-sensitive and best-effort applications concurrently on a shared infrastructure, creating resource interferences that risk QoS violations. To enforce QoS more precisely, we introduce the notion of the comprehensive latency (CL), which aggregates the following response latencies' metrics: mean, median, tail latency (99^{th} percentile), and standard deviation. Through detailed analysis, we identify CPU run-queue delays, non-voluntary context switches, cache contention, and synchronization overheads as the primary contributors to the CL degradation. To address these issues, we propose *FaaSCtrl*, a QoS-aware scheduling framework that employs reinforcement learning to dynamically adjust parameters such as the CPU core allocation and process priorities in LS applications. This design adapts to workload variability, ensuring that LS functions enjoy predictable latency guarantees while BE applications enjoy some degree of fairness and modest to high throughput.

The final challenge arises from inefficient resource utilization during the I/O stages of a function's execution. When functions interact with remote storage systems to fetch inputs or persist outputs, containers remain active but make minimal use of CPU and memory resources, resulting in significant underutilization. These idle resources, if reclaimed, could be redirected to serve new requests, thereby reducing the likelihood of QoS violations. To address this inefficiency, we propose Styx, a workflow engine that decouples compute and I/O stages during functions' execution. It incorporates a predictive prefetcher that estimates the fetch latency of a file and ensures that the input data is fetched exactly when needed, thereby reducing a request's waiting time. Additionally, it offloads the output storage phase from containers to a host-side data service, thereby reducing idle resource usage. By minimizing the provisioned memory

duration and overlapping compute with I/O, Styx improves the performance of serverless applications and reclaims memory earlier to increase throughput.

In summary, this thesis demonstrates that meeting QoS guarantees on serverless platforms requires addressing bottlenecks across the entire execution pipeline of a platform. FaaSImage accelerates image retrieval, SnapStore reduces sandbox initialization costs, *FaaSCtrl* ensures QoS compliance under contention, and Styx optimizes resource utilization. Together, these contributions span the network, disk, compute, and memory subsystems, illustrating that cross-layer resource management is essential to delivering predictable, efficient, and fair QoS in serverless computing while extending its adaptability to diverse modern applications.

सार

सर्वरलेस कंप्यूटिंग जल्दी से क्लाउड अनुप्रयोगों के लिए एक मुख्यधारा का प्रतिमान बन गया है, मुख्य रूप से फाइन-ग्रेन बिलिंग और ऑटोस्केलिंग प्रदान करते हुए डेवलपर्स को बुनियादी ढांचे के प्रबंधन से मुक्त करने की इसकी क्षमता के कारण। पारंपरिक क्लाउड मॉडल के विपरीत, सर्वर रहित प्लेटफ़ॉर्म हल्के सैंडबॉक्स में घटना-संचालित कार्यों को निष्पादित करते हैं, जिससे उपयोगकर्ता केवल वास्तविक निष्पादन की अवधि के लिए भुगतान कर सकते हैं। यह सरलता, लोच के साथ मिलकर, अमेज़ॅन, माइक्रोसॉफ्ट, गूगल और आईबीएम जैसे प्रमुख क्लाउड प्रदाताओं द्वारा इसे व्यापक रूप से अपनाने के लिए प्रेरित किया है, जो आईओटी, मशीन लर्निंग, डेटा एनालिटिक्स और बड़े पैमाने पर पोर्टलों जैसे डोमेन को शक्ति प्रदान करते हैं। हालाँकि, यह बदलाव चुनौतियों के एक नए समूह को भी उजागर करता है। सर्वर रहित कार्यभार अत्यंत विविध होते हैंः कुछ सख्त विलंबता गारंटी (विलंबता-संवेदनशील वर्ग) की मांग करते हैं जबकि अन्य थ्रूपुट (सर्वश्रेष्ठ-प्रयास वर्ग) को प्राथमिकता देते हैं। साझा बहु-किरायेदार अवसंरचना पर सेवा आवश्यकताओं की इन गुणवत्ता को पूरा करना मुश्किल है, क्योंकि सर्वर रहित निष्पादन पाइपलाइन कई चरणों में अड़चनें पेश करती है जो अनुमानित प्रदर्शन में हस्तक्षेप करती हैं। यह थीसिस चरण-दर-चरण कार्य निष्पादन पाइपलाइन की जांच करता है, ओवरहेड के प्रमुख स्रोतों की पहचान करता है और उन्हें कम करने के लिए तंत्र का प्रस्ताव करता है।

जब कोई फ़ंक्शन अनुरोध आता है, तो प्लेटफ़ॉर्म को पहले यह सुनिश्चित करना चाहिए कि उसकी डिस्क छवि निष्पादन नोड पर उपलब्ध है। अनुरोध अंतर-आगमन समय में महत्वपूर्ण परिवर्तनशीलता के कारण, छवियों के स्थानीय रूप से पाए जाने की संभावना कम होती है, जिससे प्लेटफार्मों को अक्सर उन्हें दूरस्थ रजिस्ट्रियों से पुनः प्राप्त करने के लिए मजबूर होना पड़ता है। यह छवि पुनर्प्राप्ति चरण प्रारंभिक अड़चन है, विशेष रूप से विलंबता-संवेदनशील (एलएस) अनुप्रयोगों के लिए जो अनुमानित प्रदर्शन की मांग करते हैं। इस समस्या को हल करने के लिए, हम एफएएएसआईमेज का प्रस्ताव करते हैं, एक ढांचा जो रीइंजीनियर छवि पुनर्प्राप्ति ओवरहेड्स को कम करने के लिए छवियों को कार्य करता है। एफएएएसआईमेज तीन तंत्रों को एकीकृत करता हैः लेयर सबसेटिंग, जो केवल स्टार्टअप के लिए आवश्यक फ़ाइलों को डाउनलोड करता है; ऑन-डिमांड डाउनलोडिंग, जो निष्पादन के दौरान अतिरिक्त फ़ाइलों को आलसी ढंग से पुनर्प्राप्त करता है; और लेयर फ्यूजन, जो अनावश्यक हस्तांतरण से बचने के लिए कार्यों में ओवरलैप को अधिकतम करता है। सामूहिक रूप से, ये तंत्र छवि खींचने में तेजी लाते हैं, जिससे सर्वर रहित प्लेटफार्मों में कोल्ड स्टार्ट प्रदर्शन में सुधार होता है।

एक बार छवियाँ उपलब्ध हो जाने के बाद, प्लेटफ़ॉर्म को सैंडबॉक्स आरंभीकरण करना चाहिए, जो सेवा गारंटी की गुणवत्ता को और खतरे में डालता है। उप-सेकंड निष्पादन विलंबता वाले एल. एस. अनुप्रयोगों के लिए, सैंडबॉक्स निर्माण की विलंबता प्रतिक्रिया समय पर हावी हो सकती है, जिससे सेवा उल्लंघन की गुणवत्ता हो सकती है। इस समस्या को कम करने के लिए, पूर्व कार्य ने स्नैपशॉट और पुनर्स्थापना-आधारित तंत्र का प्रस्ताव दिया, जहां एक फ़ंक्शन स्नैपशॉट एक सैंडबॉक्स की मेमोरी और सीपीयू स्थिति को कैप्चर करता है, जो बार-बार आरंभीकरण से बचकर तेजी से गर्म शुरुआत को सक्षम करता है। हालांकि, एक स्नैपशॉट का आकार सैंडबॉक्स के मेमोरी आकार के समानुपाती होता है, और उनमें से कई को संग्रहीत करने से बड़े डिस्क ओवरहेड्स का परिचय होता है, खासकर जब सैकड़ों कार्य सह-स्थित होते हैं। इसे संबोधित करने के लिए, हम स्नैपस्टोर, एक स्मृति क्षेत्र-जागरूक स्नैपशॉट डिडुप्लिकेशन फ्रेमवर्क पेश करते हैं जो एक नोड पर स्नैपशॉट घनत्व को बढ़ाता है। स्नैपस्टोर डुप्लिकेट की कुशलता से पहचान करने और हैशिंग लागत को कम करने के लिए एक प्रोग्राम के मेमोरी मैप की संरचना का फायदा उठाता है, जिससे डिडुप्लिकेशन लेटेंसी में सुधार होता है। इसके अलावा, यह डिस्क एक्सेस को अनुक्रमित करके और रैम-आधारित कैश का लाभ उठाकर स्नैपशॉट बहाली को तेज करता है। इसलिए, स्नैपस्टोर भंडारण लागत को कम करता है, बहाली में तेजी लाता है, और यह सुनिश्चित करता है कि सैंडबॉक्स निर्माण अब एंड-टू-एंड प्रदर्शन पर हावी न हो।

छवि पुनर्प्राप्ति और सैंडबॉक्स प्रारंभिककरण को संबोधित करने के साथ, अगली चुनौती निष्पादन के दौरान संसाधन विवाद है। सर्वर रहित प्लेटफ़ॉर्म विलंबता-संवेदनशील और सर्वोत्तम-प्रयास अनुप्रयोगों को एक साझा बुनियादी ढांचे पर समवर्ती रूप से निष्पादित करते हैं, जिससे संसाधन हस्तक्षेप पैदा होते हैं जो सेवा उल्लंघन की गुणवत्ता को जोखिम में डालते हैं। सेवा की गुणवत्ता को अधिक सटीक रूप से लागू करने के लिए, हम व्यापक विलंबता (सी. एल.) की धारणा का परिचय देते हैं जो निम्नलिखित प्रतिक्रिया विलंबताओं के मेट्रिक्स को एकत्रित करता हैः माध्य, माध्य, पूंछ विलंबता (99वां प्रतिशत) और मानक विचलन। विस्तृत विश्लेषण के माध्यम से, हम सी. एल. क्षरण में प्राथमिक योगदानकर्ताओं के रूप में सीपीयू रन-क्यू डिले, गैर-स्वैच्छिक संदर्भ स्विच, कैश विवाद और सिंक्रनाइज़ेशन ओवरहेड्स की पहचान करते हैं। इन मुद्दों को संबोधित करने के लिए, हम एफएएएससीटीआरएल, सेवा-जागरूक शेड्यूलिंग फ्रेमवर्क की एक गुणवत्ता का प्रस्ताव करते हैं जो सीपीयू कोर आवंटन और एलएस अनुप्रयोगों में प्रक्रिया प्राथमिकताओं जैसे मापदंडों को गतिशील रूप से समायोजित करने के लिए सुदृढीकरण सीखने को नियोजित करता है। यह डिज़ाइन कार्यभार परिवर्तनशीलता के अनुकूल है, यह सुनिश्चित करते हुए कि एल. एस. कार्य अनुमानित विलंबता गारंटी का आनंद लेते हैं जबकि बी. ई. अनुप्रयोग कुछ हद तक निष्पक्षता और मामूली से उच्च थ्रूपुट का आनंद लेते हैं।

अंतिम चुनौती किसी फ़ंक्शन के निष्पादन के इनपुट आउटपुट चरणों के दौरान अक्षम संसाधन उपयोग से उत्पन्न होती है। जब कार्य इनपुट प्राप्त करने या आउटपुट बनाए रखने के लिए दूरस्थ भंडारण प्रणालियों के साथ बातचीत करते हैं, तो कंटेनर सक्रिय रहते हैं लेकिन सीपीयू और मेमोरी संसाधनों का न्यूनतम उपयोग करते हैं, जिसके परिणामस्वरूप महत्वपूर्ण कम उपयोग होता है। इन निष्क्रिय संसाधनों को, यदि पुनः प्राप्त किया जाता है, तो नए अनुरोधों को पूरा करने के लिए पुनर्निर्देशित किया जा सकता है, जिससे सेवा उल्लंघन की गुणवत्ता की संभावना कम हो जाती है। इस अक्षमता को दूर करने के लिए, हम स्टाइक्स, एक कार्यप्रवाह इंजन का प्रस्ताव करते हैं जो कार्यों के निष्पादन के दौरान गणना और I/O चरणों को अलग करता है। यह एक प्रेडिक्टिव प्रीफेचर को शामिल करता है जो एक फ़ाइल की लाने की विलंबता का अनुमान लगाता है और यह सुनिश्चित करता है कि इनपुट डेटा को आवश्यकता पड़ने पर ठीक से प्राप्त किया जाए, जिससे अनुरोध के प्रतीक्षा समय को कम किया जा सके। इसके अतिरिक्त, यह आउटपुट भंडारण चरण को कंटेनरों से होस्ट-साइड डेटा सेवा में उतार देता है, जिससे निष्क्रिय संसाधन उपयोग कम हो जाता है। प्रावधानित स्मृति अवधि को कम करके और आई/ओ के साथ गणना को ओवरलैप करके, स्टाइक्स सर्वर रहित अनुप्रयोगों के प्रदर्शन में सुधार करता है और थ्रूपुट को बढ़ाने के लिए पहले स्मृति को पुनः प्राप्त करता है।

संक्षेप में, यह थीसिस दर्शाता है कि सर्वर रहित प्लेटफार्मों पर सेवा गारंटी की गुणवत्ता को पूरा करने के लिए एक मंच की पूरी निष्पादन पाइपलाइन में बाधाओं को दूर करने की आवश्यकता होती है। एफएएएसआईमेज छवि पुनर्प्राप्ति को तेज करता है, स्नैपस्टोर सैंडबॉक्स प्रारंभिक लागत को कम करता है, एफएएएससीटीआरएल विवाद के तहत सेवा अनुपालन की गुणवत्ता सुनिश्चित करता है, और स्टाइक्स संसाधन उपयोग को अनुकूलित करता है। एक साथ, ये योगदान नेटवर्क, डिस्क, गणना और स्मृति उप-प्रणालियों में फैले हुए हैं, जो दर्शाते हैं कि विविध आधुनिक अनुप्रयोगों के लिए अपनी अनुकूलन क्षमता का विस्तार करते हुए सर्वर रहित कंप्यूटिंग में सेवा की अनुमानित, कुशल और उचित गुणवत्ता प्रदान करने के लिए क्रॉस-लेयर संसाधन प्रबंधन आवश्यक है।

Contents

Certificate **i**

Acknowledgements **iii**

Abstract **v**

List of Figures **xvii**

List of Tables **xxiv**

1 Introduction **1**

1.1 Anatomy of a Serverless Platform 1

1.2 Overview of the Thesis 2

2 FaaSImage: An Efficient Image Manager for FaaS **9**

2.1 Introduction 9

2.2 Background 12

2.2.1 Serverless Computing 12

2.2.2 Docker 13

2.3 Characterization of a Function Image 15

2.3.1 Structure of a Function Image 15
2.3.2 Performance Analysis of an Image 17
2.3.3 Layer Fusion vs On-Demand Download 18
2.3.4 Redundancy Analysis of an Image 21
2.4 Design 21
2.4.1 Design Principles 21
2.4.2 Overview 22
2.4.3 Image Builder 23
2.4.4 Image Registry 24
2.4.5 Storage Manager 26
2.4.6 FAAS Overlay Module 27
2.5 Evaluation 27
2.5.1 Evaluation setup 27
2.5.2 Cold Start Performance 29
2.5.3 Performance Across Different Runtimes 31
2.5.4 Image Refactoring Time 32
2.6 Related Work 32
2.6.1 Layer Fusion 33
2.6.2 On-Demand Download 33
2.7 Discussion 34
2.7.1 Cluster Scale Deployment 34
2.7.2 Secure On-Demand File Retrieval 35
2.8 Summary 35

3 SnapStore: A Snapshot Storage System for Serverless Systems 37

3.1 Introduction . . . 37

3.2 Background . . . 40

3.2.1 Serverless Computing . . . 40

3.2.2 Firecracker . . . 41

3.2.3 Deduplication . . . 42

3.3 Characterization of a Function Snapshot . . . 43

3.3.1 Evaluation Methodology . . . 43

3.3.2 Analyzing Non-dirty Memory Pages . . . 44

3.3.3 Analyzing Dirty Memory Pages . . . 45

3.3.4 Analyzing Duplicate Memory Pages . . . 46

3.4 Design . . . 48

3.4.1 Design Principles . . . 48

3.4.2 High-level System Design . . . 49

3.4.3 Data Structures . . . 50

3.4.4 Preprocess Function Snapshot . . . 52

3.4.5 Region-based Backup . . . 52

3.4.6 Region-based Retrieval . . . 53

3.5 Evaluation . . . 55

3.5.1 Experimental Setup . . . 55

3.5.2 Deduplication Performance . . . 56

3.5.3 End-to-End Performance . . . 57

3.5.4 Limit Study with Different Runtime Environments . . . 58

3.6 Related Work . . . 59
3.6.1 Deduplication in Cloud Environments: Overall Operation . . . 59
3.6.2 Specific Schemes . . . 60
3.7 Discussion . . . 61
3.7.1 Metadata Management . . . 61
3.8 Summary . . . 62

4 FaaSCtrl: A Comprehensive-Latency Controller for Serverless Platforms 65

4.1 Introduction . . . 65
4.2 Background . . . 69
4.2.1 Serverless Computing . . . 69
4.2.2 Reinforcement Learning . . . 71
4.3 Related Work . . . 72
4.3.1 Resource Scheduling in the Cloud . . . 72
4.3.2 Resource Scheduling in Serverless Computing . . . 72
4.3.3 RL-based Resource Scheduling . . . 73
4.4 Characterization of a Serverless Framework . . . 73
4.4.1 Evaluation Methodology . . . 74
4.4.2 Quantifying the Interference . . . 75
4.4.3 Understanding the Interference . . . 76
4.5 Motivation . . . 78
4.5.1 Default Knob: Assigning Priority . . . 78
4.5.2 Setting the Physical CPU Affinity . . . 80
4.5.3 Effect of the Input Size . . . 81

4.6 Problem Formulation 81
4.7 Design 83
4.7.1 High-Level Overview 84
4.7.2 State Monitor 85
4.7.3 FaaSCtrl 86
4.8 Evaluation 88
4.8.1 Configuration 89
4.8.2 Analyzing Hyperparameter Sensitivity 89
4.8.3 Evaluating Comprehensive Latency and Fairness 90
4.8.4 Evaluating Scalability 92
4.8.5 Evaluating Impact of BE applications 95
4.8.6 Limit Study 95
4.9 Discussion 96
4.9.1 Cluster Scale Deployment 96
4.10 Summary 96

5 Styx: An Efficient Workflow Engine for Serverless Platforms 99

5.1 Introduction 99
5.2 Background 102
5.2.1 Serverless Computing 102
5.2.2 Container's Resource Utilization 103
5.2.3 Object Storage System 105
5.3 Characterization of a Serverless Platform 105
5.3.1 Evaluation Methodology 107

5.3.2 Metrics for Resource Utilization . . . 107
5.3.3 Function Level Breakdown . . . 108
5.3.4 System Level Breakdown . . . 110
5.3.5 Scope of Overlapping Prefetching and Initialization . . . 111
5.4 Design . . . 111
5.4.1 Design Principles . . . 111
5.4.2 Styx Components . . . 112
5.4.3 Function Registration . . . 112
5.4.4 Function Request . . . 114
5.4.5 Function Data Extraction . . . 114
5.4.6 Function Initialization and Execution . . . 116
5.4.7 Function Data Storage . . . 116
5.5 Evaluation . . . 117
5.5.1 Evaluation Setup . . . 118
5.5.2 Asynchronous Invocations . . . 120
5.5.3 Colocated Invocations . . . 121
5.5.4 Impact of an Input File's Size . . . 122
5.5.5 Impact of a Bursty Load . . . 123
5.5.6 Impact of Inaccurate *init_insts* Estimates . . . 124
5.6 Related Work . . . 125
5.6.1 Data Communication . . . 125
5.6.2 Asynchronous Data Upload . . . 126
5.7 Discussion . . . 126

5.7.1 Cluster Scale Deployment 126

5.7.2 Impact of In-Network Congestion on Prediction Accuracy 127

5.8 Conclusion 127

6 Conclusions and Future Directions **129**

6.1 Future Directions 129

Bibliography **133**

List of Publications **161**

Biography **163**

List of Figures

1.1 The overview of the thesis. 4

2.1 Layer organization and pull operation of function images: F_1 and F_2 across Docker, Gear [66], Break [67] and FaaSImage. L_1 and L_2 denote the OS and runtime; $\{L_3, L_5\}$ denote libraries; $\{L_4, L_6\}$ denote code and dependencies; and A denotes the required files for container startup. *Note: The space between layers denote the parallel fetching of layers.* 11

2.2 A system pulls a Docker image from a remote registry. Subsequently, a container uses the mount point to access files. On every access, the overlay2 storage driver searches for files in the writable layer and image (a stack of read-only layers). 13

2.3 The typical structure of a function image. It consists of two parts: the *base segment*, which denotes a set of layers that contains OS and the runtime environment, and the *app segment*, which denotes a set of layers that contains the code, dependencies, and libraries. 17

2.4 The relative contribution of layer downloading and layer untarring operations during function image pull. 18

2.5 The size of the portion of an image accessed during startup is normalized to the image size. *Note: A container accesses approximately 25.3% of the image size during startup.* 19

2.6 The startup latency of a function when using OpenFaaS + Gear normalized to that of OpenFaaS + Docker. *The startup latency increases by at most 13.5× due to on-demand download of files.* 20

2.7 Data redundancy between the libraries of function images when compared with lr_training using the file-level deduplication scheme. 20

2.8 The high-level architecture of FaaSImage. 23

2.9 The illustration of refactoring a given set of images into a common *base* layer, unique *func* layers, and shared libraries by FaaSImage. 26

2.10 The process of pulling an image in FaaSImage. 26

2.11 The cold start latency of Gear (GR), Break (BK), and FaaSImage (FSI) when images are pulled in *Config-1* (normalized to that of OpenFaaS + Docker). *Takeaway: FaaSImage improves the cold start latency by 38.7% on HDDs and 39.8% on SSDs on an average.* 28

2.12 The storage usage and download traffic experienced by a serverless node, when we execute functions in *Config-1*. 28

2.13 The cold start latency of Gear (GR), Break (BK), and FaaSImage (FSI) when function images of different runtime environments are pulled in *Config-1* on HDDs (normalized to that of OpenFaaS + Docker). *Takeaway: FaaSImage improves the cold start latency for Node.js, Ruby and PHP runtimes by 49.9%, 38.9% and 31.1% on an average, respectively.* 30

2.14 The cold start latency of Gear (GR), Break (BK), and FaaSImage (FSI) when function images of different runtime environments are pulled in *Config-1* on SSDs (normalized to that of OpenFaaS + Docker). *Takeaway: FaaSImage improves the cold start latency for Node.js, Ruby and PHP runtimes by 50.9%, 49.1% and 42.5% on an average, respectively.* 31

2.15 The time taken to convert a function image into the FaaSImage format on HDDs. 33

3.1 The high-level design of a traditional deduplication system. 40

3.2 The occupancy of non-dirty memory pages in the memory file of a function snapshot. *Note: A snapshot contains a significant amount of non-dirty memory pages that leads to storage wastage.* 44

3.3 Breakup of the dirty-memory pages in the memory snapshot file of a function. . 44

3.4 The data redundancy and total number of entries in the fingerprint table of the following regions of a serverless function: NRT, RX_MAP, and W_MAP with respect to the lr_training function by using the fixed-sized chunk algorithm and varying the chunk size from 64 bytes to 2048 bytes. *Note: The NRT and RX_MAP memory regions show high memory redundancy, while W_MAP shows low memory redundancy.* 47

3.5 The high-level design of SnapStore. 48

3.6 The deduplication performance of SnapStore where function snapshots are stored in *Config-1*. *Note: SnapStore improves the deduplication time by 50% on HDDs and 30.8% on SSDs, while limiting the deduplication ratio by 21.6%.* . . . 54

3.7 The end-to-end performance of FaaSnap along with SnapStore, MeGA and HiDeStore, where function snapshots are retrieved as per *Config-1* (normalized to that without deduplication). *Note: FaaSnap + SnapStore improves the latency by 25.9% over FaaSnap on HDDs, while it degrades by a mere 0.6% on SSDs, on average.* 56

3.8 The deduplication performance of SnapStore when storing function snapshots of different runtime environments in *Config-1*. *Note: SnapStore improves the deduplication time of a function snapshot by 51%, while limiting the deduplication ratio by 11.9%.* 58

3.9 The retrieval performance of SnapStore when storing function snapshots of different runtimes in *Config-1*. *Note: SnapStore improves the retrieval time of a snapshot by 88.71%.* 59

4.1 The components of the comprehensive latency of LS applications, when executed with LaSS (a state-of-the-art serverless platform) and Apache OpenWhisk. 67

4.2 Sensitivity of an LS application's execution delay when co-located with a BE application, both of which are running under moderate load. Each cell indicates an increase in the std. dev. or mean value of an LS application's execution latency relative to isolated execution. 75

4.3 Normalized CPU-usage counter values of the LS application (with respect to when executed in isolation) 77

4.4 Sensitivity of the execution latency of LS applications running with a priority assignment scheme when colocated with all BE applications. *Note that the RT-AP priority allocation scheme performs marginally superior than other schemes.* 78

4.5 Sensitivity of the execution latency of LS applications with varying ratios of ⟨#LS cores⟩:⟨#BE cores⟩ when colocated with the VP application, which is a BE application. *Note that the performance of OD is worst with core allocation.* 80

4.6 The σ/μ values (coefficient of variation) of the SPI metric of applications as the input size changes. 81

4.7 The high-level design of *FaaSCtrl*. 84

4.8 The working of *FaaSCtrl* 86

4.9 The average comprehensive latency of LS and BE applications when 5 random pairs of *LS* and *BE* applications are executed on a machine by changing the values of the hyperparameters: P_{step}, $a(S^{fair})$, $b(S^{perf})$ and τ using the one-at-a-time hyperparameter tuning method. 90

4.10 The components of the comprehensive latency (*CL*) of an LS application when executed along with the VP application, which is an BE application. (normalized to *isolated* execution). 91

4.11 Statistics related to the response latency of applications when an LS application is executed along with the VP application, which is an BE application. (normalized to *isolated* execution). 92

4.12 The components of the comprehensive latency (*CL*) of applications when all LS applications are executed along with all BE applications (normalized to *isolated* execution). 93

4.13 Statistics related to the response latency of applications when all LS applications are running along with all BE applications (normalized to an *isolated* execution). 94

4.14 The average comprehensive latency of LS applications when varying numbers of *LS* applications are colocated with 5 *BE* applications (normalized to their *isolated* execution). 94

5.1 A high-level overview of a traditional serverless platform executing a dummy workflow. 101

5.2 A granular view of a function's execution within a container. 103

5.3 A granular view of a function's execution with a prefetcher. 104

5.4 A granular view of a function's execution with the asynchronous data upload mechanism. 104

5.5 The percentage contribution of the init, extract, transform, store, and return (termination) stages to the execution latency of workflows listed in Table 5.1. . 108

5.6 The resource utilization of the platform when containers are in the EX active, S active, and EX+S active states during the execution of workflows listed in Table 5.1 normalized to the overall resource utilization of the platform across all container states. 109

5.7 The prefetch latency and the time spent in the *I* stage by the source functions of workflows listed in Table 5.1 that read input data from a remote storage. . . . 110

5.8 A high-level overview of Styx. 113

5.9 The fine granular view of function execution in Styx. 115

5.10 The overall memory allocated by a serverless platform while executing the serverless workflows listed in Table 5.1. Takeaway: Styx improves the memory allocation by 43.1%. 117

5.11 The tail latency and the mean latency of serverless workflows when executed on Styx and the state-of-the-art solutions (normalized to that of Pheromone + Truffle). Takeaway: Styx improves the tail latency and the mean latency of applications by 27.7% and 24.5%, respectively. 118

5.12 The tail latency and the mean latency of serverless workflows when executed concurrently on Styx and the state-of-the-art solutions (normalized to that of Pheromone + Truffle). Takeaway: Styx improves the tail latency and the mean latency of applications by 26.3% and 21%, respectively. 119

5.13 The tail latency and the mean latency of serverless workflows when the *iot* workflow executed with different input sizes on Styx and the state-of-the-art solutions (normalized to that of Pheromone + Truffle). 121

5.14 The overall memory allocated by a serverless platform while executing the *iot* workflow with different input sizes 122

5.15 The memory allocation and the latency of a serverless platform while executing the *iot* workflow with a bursty load (normalized to that of Pheromone + Truffle). 122

5.16 The memory allocation and the latency of Styx while executing the *iot* workflow with different error rates in the *init_insts* estimates under the underestimation scenario (normalized to that of Dataflower + Truffle). 123

5.17 The memory allocation and the latency of Styx while executing the *iot* workflow with different error rates in the *init_insts* estimates under the overestimation scenario (normalized to that of Dataflower + Truffle). 124

List of Tables

2.1 Workloads used in this chapter (adapted from FunctionBench [97]). *Note: The image size column denotes the size of the Docker image.* 16

2.2 System configuration 16

3.1 Workloads used in this chapter (adapted from FunctionBench [97]). *Note: The memory column denotes the size of the microVM allocated to a function.* . . . 43

3.2 System configuration 43

3.3 List of data structures used by SnapStore 50

3.4 List of runtime environments (adapted from Amazon [105]). *Note: The size of the microVM allocated for all the runtime environments is 128 MB.* 58

4.1 A summary of closely-related works to *FaaSCtrl*. 66

4.2 Workloads used in the chapter (adapted from highly cited prior work [49, 97, 192, 171]). 74

4.3 System configuration 75

4.4 Pearson's correlation coefficients of HPEs and CPU-usage events with the execution latency of LS applications. 77

4.5 State attributes of a serverless application (F_{id}). 85

4.6 Hyperparameters in the *FaaSCtrl* design. 89

4.7 The impact of different numbers of BE applications on LS applications in *FaaSCtrl* when compared against LaSS [208]. 95

5.1 The workflows used in this chapter are adopted from real-world serverless applications [97, 153, 55, 25, 22, 89]. *Note: we allocate containers of memory limit 256 MB to the face_detector, ml_normalize and ml_merge functions, while others have a memory limit of 128 MB.* 106

5.2 System configuration 107

5.3 The features and their respective data types collected from the node and the remote storage system for use by the fetch latency predictor. 114

5.4 Comparison of the fetch latency predictor with different models. 119