

UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Multi-Tenant Mobile Offloading Systems for Real-Time Computer Vision Applications

Permalink

<https://escholarship.org/uc/item/72q1420w>

Author

Fang, Zhou

Publication Date

2018

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA SAN DIEGO

Multi-Tenant Mobile Offloading Systems for Real-Time Computer Vision Applications

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Zhou Fang

Committee in charge:

Professor Rajesh Gupta, Chair
Professor Sujit Dey
Professor Sicun Gao
Professor Ryan Kastner
Professor Deian Stefan

2018

Copyright

Zhou Fang, 2018

All rights reserved.

The Dissertation of Zhou Fang is approved and is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego

2018

TABLE OF CONTENTS

Signature Page	iii
Table of Contents	iv
List of Figures	viii
List of Tables	xi
Acknowledgements	xii
Vita	xiv
Abstract of the Dissertation	xv
Chapter 1 Introduction	1
1.1 Problem Background	1
1.1.1 Mobile Offloading Techniques	1
1.1.2 Video Analytics Applications	4
1.1.3 Limitations in Existing Systems	4
1.2 Dissertation Contributions	7
1.2.1 Serving Mobile Workload in the Cloudlet	8
1.2.2 Extending the Cloudlet with the Cloud	11
1.3 Dissertation Scope	12
Chapter 2 An Overview of Mobile Offloading Systems	13
2.1 Mobile Computing Systems	14
2.1.1 Applications	14
2.1.2 Design for Resource Efficiency	15
2.2 Mobile Offloading Systems	16
2.2.1 Coarse-grained Offloading	17
2.2.2 Fine-grained Offloading	17
2.2.3 Real-Time Mobile Applications	20
2.2.4 Server Deployments	21
2.3 DNNs in Mobile Computing	24
2.3.1 DNNs for Computer Vision Tasks	24
2.3.2 Running DNNs on Mobile	25
2.3.3 Offloading DNN Tasks	26
2.4 Our Work	26
Chapter 3 DeepQuery: A Mobile Offloading Framework for Vision Applications	28
3.1 Introduction	28
3.2 Background	30
3.3 Applications	32

3.3.1	CPU Workloads	33
3.3.2	GPU Workloads	34
3.3.3	Platform Specifications	34
3.3.4	CNN Feature Extraction	34
3.3.5	Object Detection in Videos	35
3.3.6	High-Level Vision Tasks	39
3.4	Optimizing DNN Inferences	40
3.4.1	Customized Input Batching	40
3.4.2	Sharing GPUs by Multiple Models	40
3.4.3	Parallelization	42
3.5	Programming Applications in DeepQuery	43
3.5.1	Programming Model	43
3.5.2	Application Programming Interfaces (APIs)	44
3.5.3	Data Management	45
3.6	Processing Queries on Worker Servers	46
3.6.1	Processing DNN using GPUs	47
3.6.2	Processing Queries on CPUs	48
3.7	Summary	49
Chapter 4	Scheduling DNN Queries for High GPU Resource Utilization	51
4.1	Introduction	51
4.2	Background	53
4.3	Scheduling System Design	54
4.3.1	Application Timing Characteristics	54
4.3.2	Dispatching Queries to Workers	55
4.3.3	Predicting GPU Workloads	56
4.4	Scheduling Algorithms	58
4.4.1	Dynamic Batching for Parallel Stages	58
4.4.2	Dynamic Batching for Non-RT Jobs	59
4.4.3	Scaling to Multiple Workers	60
4.5	Evaluation	61
4.5.1	Dynamic Batching for Parallel Stages	61
4.5.2	Co-locating RT and Non-RT Queries	62
4.5.3	Worker Selection Strategies	64
4.6	Summary	65
Chapter 5	Mitigating Resource Contention on CPUs	67
5.1	Introduction	67
5.2	Background	70
5.3	System Model and Performance Metrics	70
5.4	ATOMS System Design	72
5.4.1	Master and Offloading Workflow	73
5.4.2	Planning Algorithms	75
5.4.3	Scheduling	79

5.5	Implementation	82
5.5.1	Measuring Computing Time	82
5.5.2	Computing Time Prediction	84
5.5.3	Upstream Latency Estimation	87
5.5.4	Clock Synchronization	89
5.6	Evaluation	91
5.6.1	Test Setup	91
5.6.2	Application Performance	92
5.6.3	Serving More Clients	94
5.6.4	Shared Worker	95
5.6.5	Service Level Agreement	97
5.7	Summary	99
Chapter 6 Data Flow-Based Mobile Offloading APIs		101
6.1	Introduction	101
6.2	Background	103
6.3	Framework Design	104
6.3.1	Application Programming Interfaces (APIs)	104
6.3.2	Queries and Services	105
6.3.3	Distributed Systems	106
6.4	Evaluation	107
6.4.1	Experimental Setup	108
6.4.2	Examining Onboard Processing and Offloading	109
6.5	Summary	110
Chapter 7 DNN Adaptive Batching for Accuracy and Delay Trade-offs		111
7.1	Introduction	111
7.2	Background	112
7.3	DNN Inference Server	114
7.3.1	System Overview	114
7.3.2	The Impact of Batch Size	114
7.3.3	Performance Metrics of Query Response	115
7.4	Scheduler Design	116
7.4.1	Simple Scheduling by Heuristic	117
7.4.2	Advanced Scheduling by Deep Reinforcement Learning	118
7.5	Evaluation	120
7.5.1	Experiment Setting	120
7.5.2	Performance of Schedulers	121
7.6	Summary	122
Chapter 8 Conclusion and Future Work		125
8.1	Conclusion	125
8.2	Limitations and Future Work	126

Appendix A Programming Customized Microservices in DeepQuery 129
 A.1 Microservices 129
 A.2 Creating Jobs for Queries 130
Bibliography 133

LIST OF FIGURES

Figure 1.1.	An example of vision analytics application using multiple DNNs.	5
Figure 1.2.	Comparing microservices deployed in the cloud and the cloudlet architectures.	8
Figure 1.3.	The outline of this dissertation.	9
Figure 3.1.	A video object detection application composed of tracking and detection tasks.	36
Figure 3.2.	Architectures of DNN models for object detection, tracking, scene graph detection and video description.	37
Figure 3.3.	DNN processing delays for object tracking and detection with and without cuDNN AutoTune.	41
Figure 3.4.	An overview of the DeepQuery system.	43
Figure 3.5.	Query processing on the worker servers in DeepQuery.	47
Figure 3.6.	An example of CPU resource contention.	50
Figure 4.1.	DNN delay estimation performance for object detection, tracking and scene graph detection tasks.	57
Figure 4.2.	CDFs of object detection and tracking delays using different batching strategies for FRCNN model.	62
Figure 4.3.	Delays of CNN and delay tail percentiles of objec tracking tasks using different batching strategies.	63
Figure 4.4.	Time slots of object detection, tracking and CNN jobs on one GPU.	64
Figure 4.5.	CDFs of object detection, tracking tasks with real-time and non-real-time scene graph detection tasks.	65
Figure 4.6.	CDFs of object detection and tracking tasks using different scheduling strategies.	65
Figure 5.1.	Demonstrating the idea of ATOMS framework.	68
Figure 5.2.	Demonstrating the performance of ATOMS framework.	69
Figure 5.3.	The architecture of ATOMS framework.	72

Figure 5.4.	Processor reservation for future offloaded task.	73
Figure 5.5.	An example of detecting processor contention.	77
Figure 5.6.	An example of coordinating reservations.	80
Figure 5.7.	Diagram of the ATOMS scheduler.	81
Figure 5.8.	Measure computing time using timestamps and CPU time.	83
Figure 5.9.	Prediction results of computing time for FaceDetect application.	87
Figure 5.10.	CDFs of errors in task computing time predictions.	88
Figure 5.11.	Upstream network latency of WiFi and LTE, with uncertainty and error of estimation.	89
Figure 5.12.	Evaluation results of Clock synchronization.	90
Figure 5.13.	Compare FCFS and ATOMS in offloading interval and E2E delay.	93
Figure 5.14.	Precision of pedestrian detecting and tracking using FCFS and ATOMS ..	93
Figure 5.15.	Offloading performance of FCFS and ATOMS running FaceDetect using WiFi.	94
Figure 5.16.	Offloading interval running FaceDetect using LTE.	96
Figure 5.17.	50% and 90% percentiles of E2E delay of FaceDetect using LTE.	97
Figure 5.18.	Offloading interval and E2E delay of ObjectDetect and FeatureMatch clients.	98
Figure 5.19.	E2E delay of ObjectDetect and FeatureMatch using different scheduling schemes.	99
Figure 5.20.	50% and 90% percentiles of E2E delay of ObjectDetect and FeatureMatch clients.	100
Figure 5.21.	Offloading interval of ObjectDetect clients using customized SLAs.	100
Figure 6.1.	An intelligent video analytics application for high-level APIs.	103
Figure 6.2.	A sample query graph using DeepVision APIs.	105
Figure 6.3.	The architecture of DeepVision.	107
Figure 6.4.	Power consumption comparison of different task placements.	110

Figure 7.1. The architecture of DNN serving system for adaptive batching. 114

Figure 7.2. Utility of query response as a function of time delay. 116

Figure 7.3. Buffering incoming queries in bins to approximately order queries 117

Figure 7.4. Taking scheduling actions at dynamic time points. 119

Figure 7.5. QPS of web queries in SogouQ dataset 121

Figure 7.6. Performance comparison of fixed, heuristic and RL schedulers. 123

Figure 7.7. The timeline plots of scheduling actions in adaptive batching. 124

LIST OF TABLES

Table 3.1.	DNN inference delays of CNN models	35
Table 3.2.	DNN inference delays of object detection models	38
Table 3.3.	DNN inference delays of object tracking models.	38
Table 3.4.	DNN inference delays of scene graph models.	39
Table 3.5.	DNN inference delays of video description models.	40
Table 3.6.	DNN inference delays of FRCNN stages.	42
Table 6.1.	DNN inference delays of the SSD-MobileNet and ResNet-50 models.	109
Table 7.1.	DNN inference throughputs of YOLO models.	115
Table 7.2.	Response quality of single GPU.	122
Table 7.3.	Portion of Tiny models of single GPU.	122
Table 7.4.	Response quality of Multi-GPUs	122
Table 7.5.	Portion of Tiny models of multi-GPUs.	123

ACKNOWLEDGEMENTS

First of all, I am deeply indebted to my advisor, Professor Rajesh Gupta, for his fundamental role in my doctoral study. Rajesh gave me encouragement to explore my interests as well as guidance to approach my goals. His continuing and enthusiastic support helped me overcome obstacles in my pursuit of the doctoral degree.

I am also grateful for my labmates at the UCSD Microelectronics Embedded Systems Lab (MESL) for their warm support and help in my study and life. My appreciations go to Omid Assare, Xun Jiao, Atieh Lotfi, Jeng-Hau Lin, Jason Koh, Vahideh Akhlaghi, Sean Hamilton, Francesco Fraternali, and Dhiman Sengupta. Many thanks to Alice Carr, the faculty assistant in our group, for her timely help on the logistic issues of all time.

I also would like to thank Professor Mani Srivastava, Fatima Anwar at UCLA, Andrew Symington at the NASA and Balkrishnan Narayanaswamy at Amazon for their enormous help to my research in the RoseLine project. Many thanks to Professor Ole J. Mengshoel, Yu Tong at CMU, Mulong Luo at Cornell, and Dezhi Hong at UCSD for helping me in various projects related to this dissertation.

I did two fruitful internship at Google in 2016 and 2017 summers. I would like to thank Dawn Chen, Yuju Hong, and Lantao Liu at the Kubernetes team, and Harsh Vardhan, Robert Bradshaw at the Cloud Dataflow team for their patient and helpful mentorship and guidance in my internship projects.

Nobody has been more important to me than my family. I would like to thank my parents and grandparents, whose love and support are always with me anytime and anywhere.

Thanks are also due to the NSF and the CONIX Research Center of the Semiconductor Research Corporation (SRC) for their financial support to my PhD study.

Chapter 3, in part, is a reprint of the material as it appears in Zhou Fang, Mulong Luo, Tong Yu, Ole J. Mengshoel, Mani B. Srivastava, and Rajesh K. Gupta. “Mitigating Multi-Tenant Interference in Continuous Mobile Offloading”, in Proceedings of the 2018 International Conference on Cloud Computing (CLOUD’18), 2018. The dissertation author was the primary

investigator and author of this paper.

Chapter 5, in part, is a reprint of the material as it appears in Zhou Fang, Mulong Luo, Tong Yu, Ole J. Mengshoel, Mani B. Srivastava, and Rajesh K. Gupta. “Mitigating Multi-Tenant Interference in Continuous Mobile Offloading”, in Proceedings of the 2018 International Conference on Cloud Computing (CLOUD’18), 2018. The dissertation author was the primary investigator and author of this paper.

Chapter 7, in full, is a reprint of the material as it appears in Zhou Fang, Tong Yu, Ole J. Mengshoel, and Rajesh K. Gupta. “QoS-Aware Scheduling of Heterogeneous Servers for Inference in Deep Neural Networks”, in Proceedings of the 2017 ACM Conference on Information and Knowledge Management (CIKM ’17), 2017. The dissertation author was the primary investigator and author of this paper.

VITA

- 2008-2012 B. S. in Microelectronics, Shanghai Jiao Tong University, China
- 2012-2014 M. S. in Electrical Engineering and Information Technology, ETH Zurich, Switzerland
- 2014-2018 Ph. D. in Computer Science (Computer Engineering) , University of California San Diego, USA

PUBLICATIONS

Zhou Fang, Jeng-Hau Lin, Mani B. Srivastava, and Rajesh K. Gupta. “Multi-Tenant Mobile Offloading Systems for Real-Time Computer Vision Applications”. Accepted for publication in Proceedings of the 20th International Conference on Distributed Computing and Networking (ICDCN’19), 2019.

Zhou Fang, Mulong Luo, Tong Yu, Ole J. Mengshoel, Mani B. Srivastava, and Rajesh K. Gupta. “Mitigating Multi-Tenant Interference in Continuous Mobile Offloading”. In Proceedings of the 2018 International Conference on Cloud Computing (CLOUD’18), 2018.

Zhou Fang, Mulong Luo, Fatima M. Anwar, Hao Zhuang, and Rajesh K. Gupta. “Go-RealTime: a Lightweight Framework for Multiprocessor Real-Time System in User Space”. ACM SIGBED Rev. 14, 4 (January 2018), 46-52.

Zhou Fang, Tong Yu, Ole J. Mengshoel, and Rajesh K. Gupta. “QoS-Aware Scheduling of Heterogeneous Servers for Inference in Deep Neural Networks”. In Proceedings of the 2017 ACM Conference on Information and Knowledge Management (CIKM ’17), 2017.

Zhou Fang, Mulong Luo, Tong Yu, Ole J. Mengshoel, Mani B. Srivastava, and Rajesh K. Gupta. “Mitigating Multi-Tenant Interference on Mobile Offloading Servers: poster abstract”. In Proceedings of the 2017 ACM Symposium on Cloud Computing (SoCC’17), 2017.

Zhou Fang, Mulong Luo, Mani B. Srivastava and Rajesh K. Gupta. “Exploiting Synchrony in Replicated State Machines”. In Proceedings of the 2017 IEEE International Conference on Cloud Computing (CLOUD’17), 2017.

ABSTRACT OF THE DISSERTATION

Multi-Tenant Mobile Offloading Systems for Real-Time Computer Vision Applications

by

Zhou Fang

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2018

Professor Rajesh Gupta, Chair

Proliferation of high resolution cameras on embedded devices along with the growing maturity of deep neural networks (DNNs) have enabled powerful mobile vision applications. To support these applications, resource-constrained mobile devices can be extended with server-class processors and GPU accelerators via mobile offloading techniques. However, this is challenging in latency-constrained applications because of large and unpredictable round-trip delays from mobile devices to the cloud computing resources. As a consequence, system designers routinely look for ways to offload to local servers at the network edge, known as the *cloudlet*.

This dissertation explores the potential of building emerging continuous mobile vision applications using the cloudlet. We identify two challenges in implementing a multi-tenant and

real-time mobile offloading framework. First, an application may exploit DNNs for diverse tasks, *e.g.*, object classification, detection, and tracking. We need methods to reduce delay and to improve throughput in DNN inference. Second, when the system hosts multiple clients and various applications concurrently, the contention on computing resources, *e.g.*, CPUs and GPUs, leads to longer delays. The scheduling techniques that mitigate resource contention are thus essential for a low-latency serving system.

To address these challenges, we design new APIs, systems, and schedulers that enable a high-performance mobile offloading framework. The framework manages data as in-memory key-value pairs that can be cached on servers to avoid redundant data transfers. An application is programmed as a Directed Acyclic Graph (DAG) of queries that process mobile data. We investigate the infrastructure optimizations, such as batching and parallelization of DNN inference, for a variety of vision tasks, *e.g.*, object detection, tracking, scene graph detection, and video description. To improve the level of resource utilization, our system co-locates delay-critical and delay-tolerant workloads on shared GPUs, using a new predictive approach. Adaptive batching algorithms that consider both accuracy and delay of DNN inference in a heterogeneous cluster of GPUs are studied as well. For CPU workloads, we present methods to mitigate resource contention by predicting future workloads, estimating contention, and adjusting task start times to remove the contention. We demonstrate the effectiveness of the framework through several evaluations with real world applications.

Chapter 1

Introduction

1.1 Problem Background

1.1.1 Mobile Offloading Techniques

High resolution cameras are becoming ubiquitous on mobile platforms such as smartphones, Google glasses [20], AWS DeepLens [19], etc. This enables a wide variety of continuous vision applications that can extract contextual information from live video streams. For example, intelligent personal assistants (IPAs) on smartphones answer queries in the form of voice and vision information to assist users [64]. Cognitive assistants on wearable glasses can also provide the environmental information to guide people with visual impairment [56]. Complementing these advances in hardware and platforms is the maturity of deep neural networks (DNNs) that are now widely adopted in computer vision applications. By quantitative measures, the canonical classes of vision tasks at the object level (*e.g.*, recognition, detection, and tracking) have achieved state-of-the-art performance [164, 65, 32]. We are also starting to see progress in tasks at higher semantic levels, such as human activity understanding [67], scene graph detection (object relationships) [155, 84], and video description [156]. The progress in platforms and algorithms is at the threshold of a greatly enhanced ability of machines to understand video content to create intelligent applications.

However, no amount of advance in computing power or efficiency can satisfy the growing needs of such applications. High performance computer vision algorithms tend to be compute-

intensive, whereas mobile devices are resource constrained. As a result, a response with a long processing delay can not provide a smooth and satisfactory user experience for iterative cognition applications [56, 37, 64]. On the other hand, continuously processing a video feed can quickly drain the device’s battery [108].

Techniques that process mobile workload remotely on server have been introduced as a generic and effective solution that is usually classified as *mobile offloading* [134, 56, 37, 60, 82]. Ubiquitous wireless networks such as WiFi and cellular networks enable highly available access to resource-rich servers. The servers can be either deployed in low-latency and high-bandwidth local clusters that provide timely offloading services, as envisioned by the *cloudlet* [134], or the cloud that provides best-effort services. There are several advantages provided by mobile offloading, compared to on-board computing on mobile platforms:

- **Low Processing Delays:** Processing computer vision tasks on server improves delay that is a key primitive of application performance. Servers are usually equipped with powerful processors (CPUs) as the computing resource, as well as accelerators including GPUs [63, 64], FPGAs [64, 51] and ASICs [78]. For example, in the evaluation of CPU workloads conducted in [37], the face detection task takes around 2s on the mobile device, whereas it takes only 0.2s on the server¹. In another experiment [82], the inference of AlexNet [86], a popular Convolutional Neural Network (CNN) model for object classification tasks, takes 81ms on a mobile GPU but only 6ms on a server-class GPU.² Although the new components of upstream and downstream network latencies are added into the end-to-end (E2E) delay, they can be minimized for latency-sensitive applications by using the cloudlet servers, which are one-hop WiFi latency away from the mobile device.
- **High Performance Algorithms:** Computer vision algorithms that attain high performance tend to be computationally intensive. Some are even infeasible for on-board computing on

¹The mobile device is a Samsung Galaxy Nexus Android smartphone, and the server is equipped with a 4-core i7 CPU [37].

²The mobile GPU is NVIDIA Kepler, and the server GPU is NVIDIA Tesla K40 M-Class [82].

mobile platforms, and can only be exploited using the mobile offloading method. Consider DNN based object detection as an example [17], the popular Faster Region-based CNN (FRCNN) [132] model using a lightweight CNN feature extractor (Inception V2 [141]) takes 58ms and attains 28% mean Average Precision (mAP) on the COCO dataset [93].³ In comparison, the FRCNN detector attains 43% mAP using the more complex NasNet CNN feature extractor [164], with an inference time as long as 1.8s even on a server-class GPU. Such high computing complexity is a barrier to use of advanced detectors on mobile platforms. Instead, these are used on remote processing resources enabled by mobile offloading.

- **Low Power Consumption:** Depending on the characteristics of computing workloads and network connections, offloading compute-intensive tasks can reduce on-board power consumption and increase the battery lifetime of mobile devices, by shifting the balance of energy consumption to an increase in communication costs while reducing high compute energy costs. For example, running an image classification on board using AlexNet consumes around twice the energy consumed when offloading the task using WiFi [82].⁴ Power consumption can be further reduced by profiling the computation and communication overheads, and making dynamic offloading decisions accordingly [112, 41, 109].
- **Simple Application Development:** The efforts in implementing mobile applications get alleviated by encapsulating the implementation inside microservices on server [130, 83]. Such approach has already been provided online by many cloud platform vendors, *e.g.*, Google Cloud Vision APIs [7] and Microsoft Cognitive Services APIs [13]. In addition, for the increasingly prevalent DNN workloads, the microservice paradigm fits well here due to the ability to accommodate evolving models, updating existing models with new training data, and managing model variants [31] provided by a DNN serving system.

³Evaluations conducted on NVIDIA GeForce GTX TITAN X GPUs.

⁴The mobile GPU is NVIDIA Kepler, and the server GPU is NVIDIA Tesla K40 M-Class [82].

1.1.2 Video Analytics Applications

Bringing together advances from computer vision, machine learning, mobile hardware and wireless networks, a rapid growth in emerging mobile vision applications is on the horizon. Many sophisticated vision algorithms, especially DNNs, can be deployed to extract rich and diverse information accurately, *e.g.*, salient objects and summary of video clips, even for latency-critical interactive applications such as IPAs [64] and cognitive assistants [56].

A representative example of such applications is shown in Figure 1.1. The application extracts visual information at different semantic levels: objects, scene graphs and video descriptions in a natural language, which can be used to guide users or answer questions in cognitive assistant applications. The vision tasks in the example can adopt DNN based methods to improve accuracy. The object tracking and detection tasks are delay-critical to capture salient objects closely. The high semantic level tasks, *e.g.*, scene graph detection and video description, are considered delay-tolerant because summarization of the video content reduces the need for real-time responses.

As another example, the performance of video description [157] can be boosted by combining 2D and 3D CNN features together with hand-crafted features (*e.g.*, improved dense trajectory [150]), extracted from raw video streams and optical flows. In doing so, a video description application involves several feature extraction tasks.

This dissertation targets mobile offloading techniques at the emerging mobile vision applications. We focus on servers deployed in a cloudlet that provide low delay [56, 37, 38] and reduce the bandwidth usage for streaming visual data to the cloud [162, 72]. Before introducing our work, we first identify key open questions in supporting such applications.

1.1.3 Limitations in Existing Systems

Recent works have made substantial progresses in optimizing mobile offloading systems, in many aspects such as end-to-end delay [134, 56, 37, 129, 65, 82], mobile power consump-

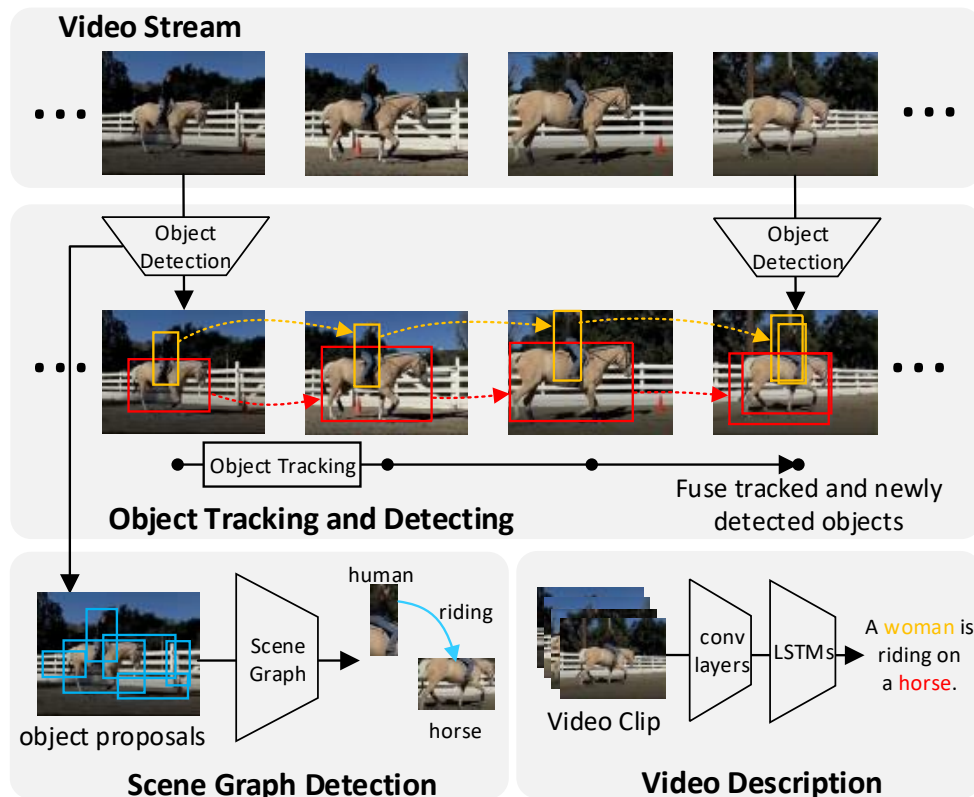


Figure 1.1. An application that retrieves visual information at different semantic levels (objects, scene graph, video description) using DNNs, from a live video stream. Our work (see Chapter 3) provides APIs to program such complex applications, with infrastructure supports to optimize DNN inference and scheduling methods to co-locate RT and non-RT tasks on shared GPUs for a higher resource utilization.

tion [41, 112, 82], and wireless network bandwidth usage [162, 37]. However, existing systems are inadequate to support the class of mobile vision applications discussed above, due to the shortages summarized here.

Complex Vision Applications: A vision application usually comprises several data processing modules including pre-processing (video ingestion, decoding, *etc.*), vision algorithms and post-processing. The modules may have data dependencies between each other. Different applications may also share common processing modules. However, existing mobile offloading frameworks usually consider simple tasks offloaded individually, such as object recognition

or detection tasks [56, 37]. A simple offloaded task is usually implemented using a remote procedure call (RPC): it sends data to the server and receives results back [56, 60, 82]. The lack of support for composite applications is clearly felt in the missing functionalities, *e.g.*, scalability, configuration knobs for resource-performance trade-off, workload scheduling techniques. In addition, applications consisting of multiple tasks may have data dependency between tasks, for example, object tracking and scene graph detection tasks depend on the outputs of object detection in Figure 6.1. This simple offloading approach is inefficient since intermediate data have to be transferred between the client and the server, which is unnecessary.

As a remedy, easy-to-use APIs to build the application and an underlying distributed system to deploy processing modules as microservices can improve the efficiency of offloading more complex applications. Although vision analytics platforms [160, 100, 72] present APIs and systems of this type, these target at cloud-scaled video analytics workload for stationary cameras. Essential issues such as the support for sub-second level real-time (RT) workloads, integration with mobile computing platforms and executing DNN models on GPUs remain untapped currently.

Multi-tenancy: The ability to serve multiple clients using different applications concurrently is critical to a mobile offloading system. The framework should support multi-tenancy to manage connections, data and offloaded tasks from different clients. Under this multi-tenant setting, contention on a variety of resources such as CPU, GPU and network bandwidth may take place, leading to the degradation of application performance. However, existing offloading frameworks are usually evaluated in a single-tenant setting [56, 37, 60, 82]. The multi-tenancy issue has not received enough attention despite of its significance.

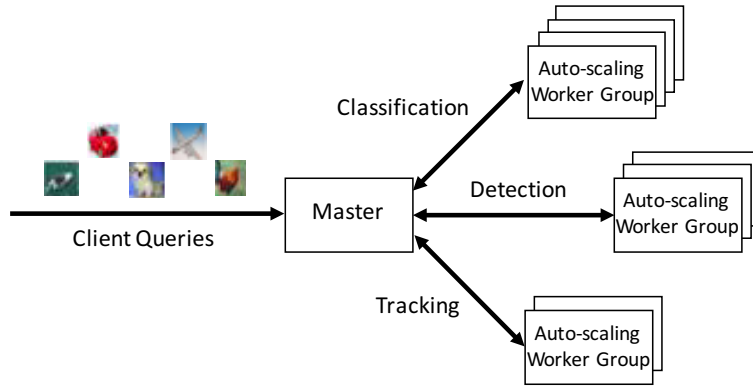
Low-latency Serving on Cloudlet: Compared to the cloud, a cloudlet has several advantages in providing low-latency services. Sub-second level delays of RT mobile tasks can be achieved by using low-latency, one-hop, high-bandwidth WiFi networks connected with local cloudlet servers [56, 37]. It also avoids streaming a large amount of data over the public network to the cloud, by aggregating and processing data at the edge servers [162].

Workload scheduling problems have been thoroughly studied for big data systems in the cloud [68, 147, 114, 146, 127]. However, unlike the cloud which is on a datacenter-scale, computational resources in the cloudlet are limited, and therefore interim shortage of resources may take place. As a result, workload scheduling in the cloudlet is quite different from the cloud. For example, as illustrated in Figure 1.2, a simple yet widely adopted method to handle a dynamic amount of incoming low-latency queries is to use an auto-scaling worker group for each microservice [1, 11], which is automatically scaled up when the resource utilization (*e.g.*, CPU, memory) exceeds a predefined threshold. Low query processing delays can be guaranteed by keeping the server utilization level low in this way: the server utilization on Amazon EC2 was estimated to be around 3% to 17% [95]. However, for a cloudlet with a limited number of worker servers, it shares workers between different microservices to improve resource utilization. In this case, fine-grained scheduling techniques that decide when, where and how long to process each workload is essential to optimizing its performance, which are explored in this dissertation.

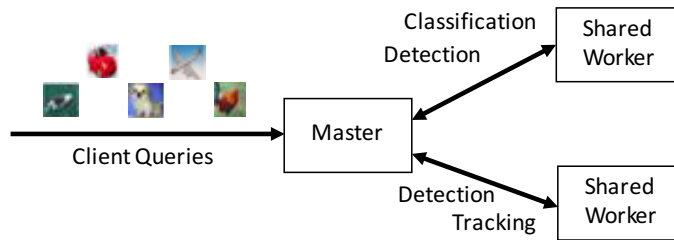
There has been a large body of research on implementing more sophisticated schedulers in the cloud. For example, previous works provide methods to improve resource utilization [68, 147], satisfy data processing jobs with timing requirements [146, 127], as well as to schedule low-latency iterative queries [114, 28]. The schedulers presented in our work use similar ideas in server load monitoring, workload execution time prediction, and load-aware workload allocation. As the novel capabilities of our schedulers, we present methods to mitigate resource contention to the maximal extent to provide low-latency performance on heavily loaded local servers (Chapter 4 and 5).

1.2 Dissertation Contributions

This dissertation presents a comprehensive study of the design of a low-delay, high-performance mobile offloading framework on a cloudlet, for serving computer vision applications that heavily leverage DNNs. Figure 1.3 outlines this work. To address the above issues, we



(a) Microservices using auto-scaling worker groups in the cloud.



(b) Microservices using shared workers in the cloudlet.

Figure 1.2. Comparing microservices deployed in the cloud and the cloudlet architectures. For the cloud with dynamically provisioned worker servers, an auto-scaling worker group can be used to maintain a low resource utilization to provide low-latency responses [1, 11]. However, for the cloudlet with limited resources, workers may be shared by a few microservices, which needs more sophisticated scheduling methods.

designed novel programming APIs to build applications (Chapter 3), a serving system optimized for executing DNN models (Chapter 3), and scheduling algorithms that mitigate multi-tenant resource contention (Chapter 4, 5). We further extended the computing resources by studying high-level APIs that distribute computation across mobile devices, cloudlet and cloud servers (Chapter 6), as well as adaptive batching methods for DNN serving systems (Chapter 7).

1.2.1 Serving Mobile Workload in the Cloudlet

Designing APIs and distributed systems

We designed a set of new APIs to program offloaded applications as directed acyclic graphs (DAGs) of individual tasks. The APIs, together with the underlying distributed system

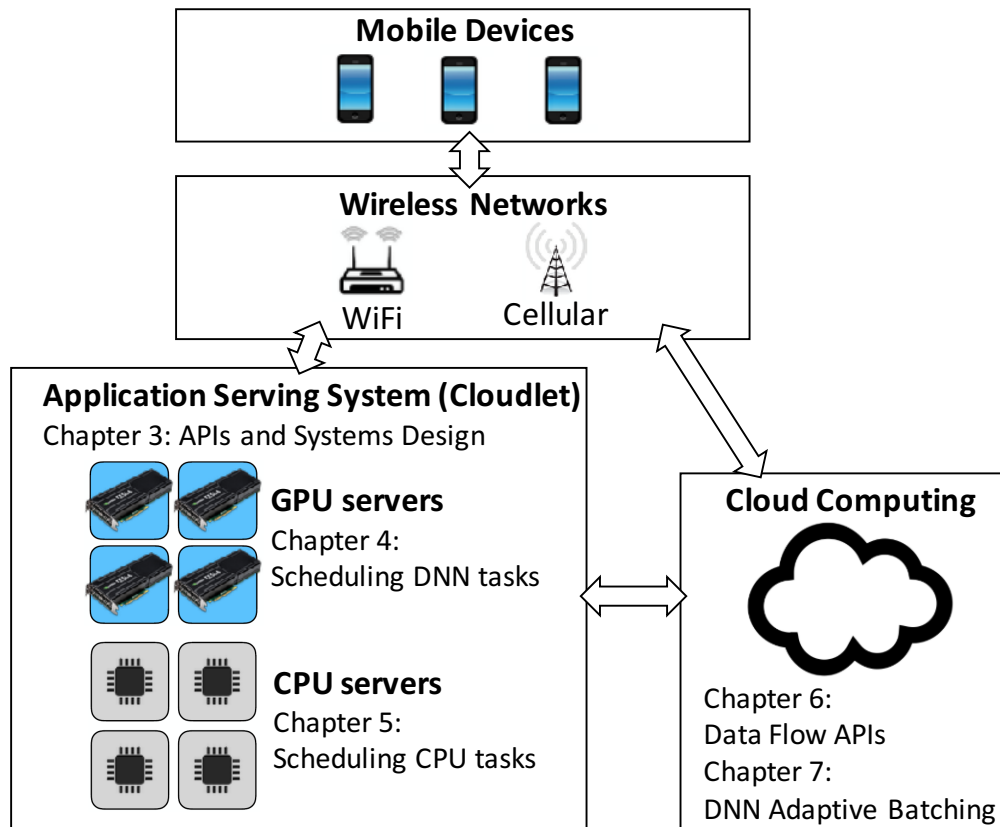


Figure 1.3. The outline of this dissertation. We present the implementation of *DeepQuery*, a mobile offloading system for a cloudlet [134] in Chapter 3. The scheduling algorithms for DNN inference tasks on GPUs and computer vision algorithms on CPUs are described in Chapter 4 and 5 respectively. Then we extend local servers by considering cloud servers as well. A mobile offloading system using a data flow model is presented in Chapter 6. DNN adaptive batching algorithms are studied in Chapter 7.

that processes tasks, are implemented in *DeepQuery*, a mobile offloading framework presented in Chapter 3. It manages a cluster of machines, and deploys algorithmic modules, *e.g.*, DNNs, as microservices running inside individual Linux containers. Data are managed as in-memory key-value pairs. Vision tasks are queries that take keyed data as the input. The APIs support fine-grained data management, such as caching data, on the server side. It effectively avoids redundant data transfer between client and server, and shares queries by different applications to reduce computational overhead.

Executing DNN Tasks on GPUs

We made the following contributions that address a host of new challenges in serving DNN models for real-time mobile applications.

- **Optimizing Diverse Models:** In addition to Convolutional Neural Networks (CNNs), which are the primary vision workload considered in our work [63, 40], our system supports a wide range of DNNs for diverse tasks with distinct network structures. We implemented specific infrastructure supports to optimize DNN inference, for instance, data parallelization for low delay and query batching for high throughput. These techniques are presented in Chapter 3.
- **Efficiently Using GPU Resources in the Cloudlet:** Running complex vision applications using DNNs consumes a large amount of GPU resource. On a cloudlet whose GPU resource may be limited, there will be resource shortage if GPU utilization must be kept low on each machine to provide low-delay services. In view of the fact that many vision tasks are delay-tolerant (non-RT), *e.g.*, tasks on higher semantic levels such as video description, co-locating RT and non-RT tasks improves resource utilization. The lack of support for task preemption on GPUs is an obstacle to exploiting task co-location, because non-RT tasks must yield resource to RT tasks. To solve the problem, we designed a novel *predictive* and *planning-ahead* based scheduler to reserve GPU resources ahead for RT tasks, as presented in Chapter 4.

Mitigating Resource Contention on CPUs

In multi-tenant systems, for offloaded tasks running on CPUs, contention for shared computing resources can unfortunately result in delays and application malfunction. To attack the problem, we designed a *Plan-Schedule* approach that coordinates future tasks to remove or reduce potential contention, and then dispatches incoming tasks to minimize resource contention, based on the current workloads on each server machine. The task coordinations ahead of time

are based on accurate workload predictions through processor resource modeling, computing time prediction and network latency estimation. We implemented the methods into a framework called *ATOMS* (Accurate Timing prediction and Offloading for Mobile Systems). The system uses estimation of wireless network delays and uncertainties, as well as client-server clock synchronization techniques, to get accurate timing information of offloaded tasks, for scheduling purposes. The results show that the Plan-Schedule algorithm is able to maintain low end-to-end delays even when the system utilization is very high (80%). The scheduler for CPU workloads named ATOMS, featuring workload coordination to mitigate multi-tenant interference, is presented in Chapter 5.

1.2.2 Extending the Cloudlet with the Cloud

In addition to executing vision workloads on the cloudlet, we also considered extending our framework by running workload on mobile device and servers deployed in the cloud. Chapter 6 presents a set of high-level APIs that partition a mobile vision application across mobile devices, the cloudlet and the cloud. In Chapter 7, we studied the adaptive batching problem of serving DNN queries on a heterogeneous cluster of GPU devices, which fits in both the cloudlet and cloud. We designed adaptive batching algorithms considering two Quality of Service (QoS) metrics for query responses: *accuracy* and *delay*. The algorithms dynamically select a batch size and a DNN model to process on a GPU device, with the goal to maximize the serving performance that is an utility function of inference accuracy, delay, and query deadline. A simple heuristic algorithm and a more advanced algorithm based on deep reinforcement learning (RL) are presented. The RL scheduler is found to improve the server performance measured by customized response quality functions, through automatically switching to faster DNN models when query rate levels are high.

1.3 Dissertation Scope

This dissertation studies mobile offloading techniques for computer vision applications with focus on servers deployed in the cloudlet. The APIs to efficiently build offloaded vision tasks, as well as the systems to process vision tasks, including service deployment, workload scheduling and time synchronization methods, are presented in detail. The optimization techniques to improve the runtime efficiency of DNN inference tasks are explored as well, such as data parallelization and adaptive batching in the inference phase of DNN models. We implemented several real world vision applications to evaluate our systems.

We employed a few advanced computer vision algorithms and DNN models to implement mobile applications. However, the design and training of these models are out of the scope of the dissertation. In the system aspect, we assume that there is sufficient wireless network bandwidth provisioned for offloading tasks and streaming video feeds. This work focuses on improving resource utilization levels on small clusters such as the cloudlet. Datacenter scale workload scheduling issues, such as load balance and admission control, are out of our scope.

Chapter 2

An Overview of Mobile Offloading Systems

In this chapter, we give an overview of the state-of-the-art mobile computing systems. The overview introduces both on-board computing and offloading techniques, for applications using different type of mobile sensory data, with a focus on computer vision applications using DNN methods.

The recent advancement in sensing capabilities on mobile devices has made many interesting applications possible. By analyzing data obtained from sensor devices such as camera, microphone, accelerometer and GPS, mobile applications can extract useful information and assist users in many different ways, for instance, recognition assistance [56, 38] and augmented reality applications [77, 81] that process streaming video data; activity recognition and behavior monitoring applications that leverage motion data [122, 107, 123], and acoustic event detection using audio data [53, 113]. Continuous sensing applications can require intense computational power on mobile devices because they usually leverage machine learning algorithms that may be compute-intensive. However, no advance envisioned for future in computing power or efficiency can satisfy the growing needs of such applications. To address the issue, the mobile computing community research has taken two directions:

- **On-board Computing:** One direction is to develop simple and efficient data processing algorithms for mobile platforms [117, 91], as well as design scheduling algorithms to

improve mobile resource utilization levels [33].

- **Mobile Offloading:** The other direction is to offload workload to computationally powerful servers [41, 134, 55, 121, 161, 136, 105, 75], which is enabled by the ubiquity of network connections on mobile devices and the accessibility of remote services deployed on servers.

In the rest of this chapter, we describe mobile sensing applications and on-board processing techniques in Section 2.1. Mobile offloading systems are discussed in Section 2.2. Mobile application using DNN methods are described in Section 2.3. Workloads with low-latency requirement, especially those for computer vision applications, are introduced in Section 2.2.3.

2.1 Mobile Computing Systems

2.1.1 Applications

Driven by the advances in both hardware and data processing algorithms, the rich set of sensors equipped on modern mobile devices enables many emerging applications as described below:

- **Camera:** High quality cameras are ubiquitously available on today’s mobile devices. Many visual perception applications process real-time video stream provided by camera, for instance, augmented reality [77], object detection and tracking [37], visual cognitive assistance that assists visually impaired people [56, 163], and crowd-sourced video collection on wearable devices [137].
- **Microphone:** Audio data sampled from embedded microphones on mobile devices are used in sound-related contextual inference tasks. For example, Auditeur [113] is an acoustic event detection platform for smartphones. It classifies acoustic events using energy efficient algorithms. The DSP.Ear framework [53] estimates the number of speakers and identifies the speakers.

- **Accelerometer:** On-board accelerometer is another class of ubiquitously equipped sensors on mobile devices. It generates data for applications to extract information related to activity detection [88], user behavior monitoring [122], social situation recognition [107, 123], and indoor localization [143].

The above applications process streaming data continuously to retrieve the latest information. The data processing algorithms are usually based on machine learning techniques and can be compute-intense, especially for computer vision applications that use deep learning methods. Therefore, it is crucial to develop mobile computing frameworks to efficiently execute the applications.

2.1.2 Design for Resource Efficiency

Efficient and continuous processing of data on mobile platforms depends upon the following strategies currently in use regarding resource management:

Workload Management: Continuous sensing applications put recurring computing stress on mobile devices. Because OS scheduling that addresses fairness does not take into consideration the application specific information, it may result in resource contention that degrades the performance of background sensing tasks, as well as latency-sensitive front-end applications for interaction with users. SymPhoney [79] is a framework that coordinates concurrent contending applications, to maximize the utility under constraints on resources. On demand adaption that changes processing rate on microphone and accelerometer data based on input data quality and human behavioral patterns is proposed in Jigsaw [99], as a solution to improve resource efficiency. Considering that many applications may process common sensor data, DataBank manages the delivery and operation of data to reduce the redundancy in data passing and processing [80].

Algorithm Simplification: Computational overhead can be diminished by using simplified algorithms, at the cost of possible performance degradation. For example, computer vision algorithms can be made faster by tuning parameters such as the sliding window size in object

detection [42], or the number of feature points in stereo reconstruction [98, 30]. Approximate computing is one such method in this class. To import approximate computing into mobile platforms, MobiDiC [117] offline determines the approximable tasks in an application and selects a suitable approximate version of the tasks at runtime. For DNNs, a great amount of research efforts are devoted to execute models with lower delay and less power consumption, mainly through various model simplification techniques [59, 58, 153]. We introduce these works in Section 2.3.2.

External Hardware: External on-board hardware can share the computational burden put on mobile processors. Consider a few examples next, the DSP.Ear framework [53] processes data on low-power DSP co-processor in commodity mobile devices. Running low-power DNNs on DSP on mobile devices is studied in [91]. MobileHub [135] automatically rewrites applications to utilize *sensor hub*, a micro-controller unit that integrates data from different sensors and processes them without additional effort on re-programming, and dynamically decides whether using sensor hub changes the application semantics.

Although computing efficiency can be improved, on-board computing is still limited for the workloads that are too computationally expensive for mobile platforms. In addition, performance loss due to algorithm tailoring may not always be acceptable for many applications. In the rest of this chapter we introduce mobile offloading techniques that aim to address the issue.

2.2 Mobile Offloading Systems

A mobile offloading system provides the capability to process workloads remotely on servers. According to how an application is partitioned across the mobile device and the server, mobile offloading frameworks can be classified into two classes: coarse-grained [56, 137, 60] and fine-grained offloading frameworks [112, 41, 55, 83, 109, 121].

2.2.1 Coarse-grained Offloading

Coarse-grained offloading encapsulates an entire application or a well-defined processing stage as a service provided by the server. The server-side code runs inside a virtual machine (VM) or Linux container. The complexity of deployment, including serving applications with diverse programming languages, OSs, runtimes and dependencies, is thus mitigated. The service can be provided through HTTP or RPC interfaces that have a lightweight stub on the client side.

Gabriel [56] is a cognitive assistance system for wearable devices. It offloads real-time video stream to infrastructure and run various cognition applications (e.g. face recognition, object recognition). Each application is encapsulated in a VM instance. A controller VM instance is responsible for forwarding the video stream to multiple VMs. Humans are very sensitive to the perception delay. Therefore, Gabriel has a tight bound on the end-to-end latency of the applications. It achieves low latency by offloading computation to the cloudlet [134]. GigaSight [137] is another framework that leverages cloudlets to continuously process real-time video stream. It collects large scale crowd-sourced videos from devices such as Google glass. Glimpse [37] offloads object detection tasks to achieve real-time trackability. MCDNN [60] runs multiple DNNs simultaneously on incoming high-datarate sensor streams, *e.g.*, video, audio, depth and thermal video. The framework serves DNN instances on the server, and adopts optimized DNN models that consume less resources. It uses less accurate variants of optimized models while serving high workloads.

2.2.2 Fine-grained Offloading

Fine-grained offloading frameworks partition an application between the mobile device and the server to optimize a few performance metrics, such as energy consumption, processing latency, and throughput. The advantage of fine-grained offloading is that an optimal offloading schedule can be devised, by answering questions such as whether to offload or not, which part of code and when to offload.

Implementing Fine-Grained Offloading

Fine-grained offloading frameworks invoke a subset of the methods of an application instead of offloading the entire application. To facilitate server-side method invocation, the frameworks automatically generate application code to be called [41, 83, 112, 121] or migrate thread to execute application code [55, 39].

Among the examples of frameworks that generate code, MAUI [41] is designed to minimize the energy cost of offloaded applications. It leverages code portability on managed code environment (Microsoft .NET Common Language Runtime) to create a copy of the application executable on servers. It uses proxies on the mobile and server side to transfer state that is needed by remote execution. ThinkAir [83] provides method-level computation offloading for applications on smartphones. It has a Remotable Code Generator that generates method wrappers and utility functions to generate the runtime for remote execution. Wishbone [112], a system that statically partitions applications between sensor nodes and servers, adopts a high-level stream-processing language that constructs a dataflow graph of stream operators. Odessa [121] is a framework to run perception applications processing real-time video stream. It adopts a parallel processing framework to hide the complexity of parallel and distributed programming, such as migrating stages of a processing pipeline between machines.

Thread migration techniques transparently offload unmodified mobile applications. COMET [55] transparently migrates Java thread using Distributed Shared Memory (DSM) techniques. It synchronizes the states of virtual heap, stacks, bytecode sources, class initialization states, and synthetic classes, between the mobile device and the cloud. CloneCloud [39] automatically transforms mobile applications for offloading. It partitions the application across the smartphone and the server, and then instruments the methods to be offloaded with VM instructions for migration and re-integration. To migrate a thread, the state (virtual state, program counter, registers, and stack) is packaged and shipped.

Application Partitioning

In fine-grained offloading, code to be executed on the server is decided statically or dynamically based on the execution time, network latency, power consumption, and the budget of cloud resources. In Wishbone [112], the profiling of applications is done offline using simulation or real hardware, which measures the overhead of data processing operators. The partitioning algorithm aims at minimizing both network bandwidth and CPU load on embedded systems according to profiling data. The optimization process is modeled as solving an integer linear programming (ILP) problem. A static partition may become sub-optimal when the network connectivity (*e.g.*, delay, bandwidth, packet loss rate) or computational overhead changes at the runtime. CloneCloud [39] improves the performance of offline partitioning by launching the most suitable one from a database of pre-computed partitions, according to measured current execution conditions (*e.g.*, availability of cloud resources, network connectivity). Dynamic profiling and partitioning techniques perform better in such highly varying conditions. It has been adopted in many frameworks that adapt to different goals and constraints [41, 39, 109, 121]. In the MAUI framework [41], the methods that run remotely are selected at the runtime based on the measured power consumption, application characteristics and network connectivity.

Odessa [121] profiles the application online periodically and does not require prior runtime information. Instead of solving an ILP problem, it uses a greedy and incremental algorithm to devise the decisions on offloading and level of parallelism for stages in the processing pipeline. Besides cloud and cloudlet, the ORBIT system [109] partitions code among the tiers of a smartphone, an energy-efficient peripheral board¹, and a cloud service. It partitions tasks to minimize system energy consumption with deadline constraints by the sensing applications, and periodically refines the partition. LEO [54] is a scheduler designed to maximize the performance for multiple continuous mobile sensor applications, by distributing tasks on CPU, co-processor, GPU and the cloud. The scheduling process runs on a low-power co-processor unit (LPU) to

¹For example, the Arduino [18] and IOIO [21] boards.

reduce power consumption.

2.2.3 Real-Time Mobile Applications

E2E delay is an important performance metric for mobile offloading systems. Since many applications are interacting with users, responses that arrive with low delays improve user experience. Cognitive applications usually target at sub-second E2E delays to provide similar experiences as tasks conducted by human [56]. For example, a previous experimental study takes human subjects 370ms to recognize familiar faces and 620ms to recognize unfamiliar faces [125].

To provide low delays for offloaded tasks, Gabriel [56], a cognitive assistant system, deploys computing engines in a nearby cloudlet that are only one WiFi hop away, in order to minimize network latency. More latency-critical cognitive applications on Gabriel are evaluated in [38]. It leverages multiple models to deliver the earliest result that satisfies accuracy requirement to reduce E2E delay. Glimpse [37] is a real-time object tracking framework that achieve high the tractability of object tracking in an offloading setting. Glimpse separates the application into a recognition task and a tracking task. The compute-intensive object detection task runs on the server side, whereas the lightweight tracking task runs on the mobile side more frequently. The tracking tasks running on the client side use stale results from the server to hide network delays.

Timecard [129] provides consistent end-to-end server response time for mobile applications. This task is challenging because many components have to be created and combined for each response, thus it has a highly varying execution time. Moreover, the end-to-end latency consists of the components of upstream and downstream latencies, which depend on the network connectivity and the payload size. As a best-effort approach, Timecard measures the elapsed time since the beginning of the request, and predicts the remaining time budget for request processing, according to the latency constraint. The services provided by server are designed to adapt for a given latency and quality of result trade-offs. It speeds up the processing when the remain time

budget is not enough. Timecard runs time synchronization between the mobile and the server to eliminate the clock offset.

2.2.4 Server Deployments

The offloading servers can be deployed in cloud environments, or a cloudlet setting where mobile clients connect to servers using high-bandwidth WiFi networks [134, 56]. The server deployment to choose depends upon the application requirements and workload characteristics.

Servers in Datacenter: the Cloud

Cloud computing infrastructure vendors provide highly available, on-demand resource-rich servers. Many mobile applications have already extensively utilized the computing resources in the cloud. For example, IPAs that assist users through natural language and image interaction have a majority of workload processed on the cloud servers [63]. The advantage of cloud servers is the pool of rich computational resources. A datacenter contains a large number of servers with powerful CPU processors, and accelerators including GPUs [63, 64], FPGAs [64, 51] and ASICs [78]. The server can thus handle requests from a large group of users. In addition, the cloud server vendors are responsible to ensure the services are running correctly. It thus alleviates the burden of application developers to maintain their remote services.

The lack of support for low-latency services is one of the disadvantages of cloud servers. A survey on RT virtualization techniques and predictable cloud computing in [52] has summarized the shortcomings. First, the performance metrics and Service Level Agreements (SLAs) of RT computing can not be directly translated to cloud computing SLAs. More specifically, the timing requirement of RT applications given in E2E delay is hard to translate into capacity requirements for machines and their networks. Second, RT scheduling of VMs, such as scheduling algorithms in the hypervisor layer, is needed to provide guaranteed performance. Moreover, the Quality of Service (QoS) of communication networks, *e.g.*, wireless cellular networks, wired public IP networks, is not guaranteed for their latency and bandwidth. As a result, the latency

of mobile offloading request at the server side may have large uncertainties that degrade the application performance.

Local Servers: the Cloudlet

An alternative is to deploy servers in local trusted clusters of machines known as the Cloudlet [134]. Compared with cellular networks and public IP networks, network delays in a cloudlet setting get effectively reduced by using low-latency, one-hop, high-bandwidth wireless accesses to servers. The latency constraints for RT mobile tasks can thus be achieved. It has been adopted in mobile frameworks for real-time applications [56, 37] that require end-to-end delays on a sub-second level.

Compared to the cloud, the disadvantage of local servers is that the amount of computational resource is still limited to serve a group of concurrent users. Unlike lightweight web requests, continuous sensing applications may induce high utilization on servers. Adding machines dynamically may be not feasible for the cloudlet that is not as resourceful as the cloud. Therefore, common approaches of maintaining low latency in the cloud, such as admission control and auto-scaling based on processor utilization [1, 11], are found with several shortages in this case. Efficiently utilizing local resources are crucial to support multiple clients on the cloudlet, especially for RT applications, which is the focus of our work.

Workload Scheduling

Our contributions on workload scheduling in the cloudlet are closely related with the big data frameworks and serving systems in the cloud [68, 147, 114, 146, 127]. In the systems designed for data analytics jobs in a large cluster, *e.g.*, MapReduce [45] as the most representative example, a cluster-wise daemon collects the information of available resources on worker machines, and allocates the resources to multi-tenant workloads. TetriSched [146] leverages deadline and estimated execution time information to schedule jobs ahead of execution: It decides whether to wait for preferred resources that are currently busy. It is proven to achieve

both high Service Level Objective (SLO) attainment and resource utilization. They schedule batch data processing jobs with makespan and deadline usually ranging from seconds to hours. Sparrow [114] considers low-latency iterative queries that result in a much higher throughput of scheduling actions. It implements a decentralized scheduling method using *batch sampling* to peek job queue length of multiple workers, and select the least loaded workers. For heterogeneous workloads involving both low-latency and throughput-oriented jobs, an architecture that queues jobs on each worker server is explored in [127] in addition to a global job queue. It implements queue sizing, reordering, and job to queue allocations to obtain accurate workload placement.

This work considers a similar problem in scheduling workloads for both high resource efficiency and low delay. However, we consider computer vision and DNN workloads, instead of big data computation and queries. The scale of our local clusters are much smaller than the cloud, therefore we use centralized schedulers that use accurately estimated server load and workload execution time. In addition, this work focuses on contention mitigation techniques to attain low delays (Chapter 4 and 5), which are different from the resource allocation [68, 147, 146], workload allocation [114], and queue management [127].

Serving Applications

Serving systems usually deploy applications in running environments such as VMs and containers, to hide the complexities of diverse programming languages, runtime systems, and dependencies. It also has the benefits of simplifying application deployments, *e.g.*, creating, restarting and scaling-out services.

Virtual Machines: VMs are adopted by many mobile offloading servers [56, 136, 57]. A VM encapsulates each application in an individual OS on emulated computer systems to separate it from others. It is important to keep the suitable number of VMs with the correct capacities, for purposes of both satisfying client demand and saving monetary cost. Dynamically provisioning VMs greatly simplifies deployments of offloading servers. COSMOS [136] is a middle layer between the mobile applications and the cloud servers that automatically manages

VM instances. It uses a heuristic algorithm to select the type and amount of VM instances, to ensure that enough resource is provided while minimizing the total cost. Rapid just-in-time provisioning of cloudlets is described in [57]. The problems that involve deploying new cloudlets for load balancing, hardware upgrade, or recovery from disaster are studied.

Linux Containers: Linux container is a virtualization method to run multiple isolated container environments on top of a single Linux kernel. Compared to VMs, containers get rid of the overhead of virtualizing the whole OS and attain equal or better system performance [49]. The fast advancement in container management tools, such as Docker [5] and Kubernetes [10], greatly facilitates the process to deploy a cluster of containers. Because of these advantages, it is increasingly popular to deploy microservices using container. For example, Cloudpath [110] and LAVEA [158] leverage serverless containers to run lightweight stateless application functions across hierarchical clusters. Clipper [40] deploys each DNN model in its own lightweight container that communicates with the master server through RPCs, to minimize the system overhead.

2.3 DNNs in Mobile Computing

There has been a rapid growth in deploying DNNs on mobile platforms for their versatility and demonstrated value. This section discusses computer vision tasks that use DNN models, as well as mobile computing and offloading systems designed for DNN workloads.

2.3.1 DNNs for Computer Vision Tasks

The past few years have seen an explosion of vision applications driven by DNNs. On one hand, the models for object level tasks get greatly improved, which contain the canonical computer vision tasks such as object recognition, detection and tracking. To name a few, object recognition using NASNet [164] has achieved 96.2% top-5 accuracy on the ImageNet dataset [46]. Mask R-CNN [65] has achieved 37.1% mAP in instance segmentation tasks, and 39.8% mAP in object detection tasks on the COCO dataset [93]. Many applications leverage

on these tasks. For example, cognitive assistant on mobile devices [56], object detection and tracking for auto-piloting [154], pedestrian identification on surveillance cameras [116]. On the other hand, substantial progresses have been made for tasks on higher semantic levels, such as human activity understanding [67], object relationship detection [85] and video description [156]. These advances greatly enhances machine’s capability of understanding vision data and delivers emerging cognitive applications on mobile devices.

2.3.2 Running DNNs on Mobile

Methods to simplify and accelerate DNN model in inference phase have attracted intensive research interest. Designing and training a lightweight model, which has fewer layers and parameters, effectively reduce the amount of computation. For example, SqueezeNet [74] and MobileNet [70] are lightweight models for object recognition tasks. For object detection tasks that are more complex, SqueezeDet [152] present a lightweight DNN architecture for self-driving tasks. General purpose detectors, *e.g.*, SSD [97] and FRCNN [132], can use lightweight CNNs as feature extractors to achieve real-time performance. However, lightweight models come with performance degradation. The speed and accuracy trade-offs for object detectors give good examples, which have been thoroughly discussed in [71]. The methods that prune DNN models [59, 58] remove neurons or weight connections while maintaining as much as possible the model performance. The quantization of model weights from floats to fixed point numbers can also accelerates model inference [153], especially on specific hardwares such as FPGAs and ASICs.

Many frameworks have considered DNN optimization specifically for mobile platforms. DeepX [90] is a software accelerator for deep learning algorithms. It decomposes DNNs into unit-blocks to be efficiently executed by heterogeneous on-board computational resources. SparseSep [33] reduces the computational overhead of DNNs by leveraging the sparsification of fully connected layers and separation of convolutional kernels. Optimizing DNN inference according to user needs is another way to reduce resource overhead. Pervasive CNN (P-CNN) [139]

presents a CNN inference framework driven by user satisfaction. It compiles optimal kernels based on the requirement of users offline, then tunes its accuracy during a runtime management phase. AdaDeep framework [96] explores the trade-offs between performance and resource constraints by user-specified needs. MCDNN [60] present the Approximate Model Scheduling technique that systematically trades off DNN classification accuracy for resource use.

Specifically for vision applications, DeepMon [73] presents method to reuse the intermediate results of the previous frame in a continuous video to reduce the amount of computation, using a smart caching mechanism. DeepEye [104] presents an inference software pipeline to process vision models by interleaving the execution of computation-heavy convolutional layers with the loading of memory-heavy fully-connected layers on a commodity wearable processor.

2.3.3 Offloading DNN Tasks

Similar to the issues that we discussed in 2.1.2, offloading DNNs has several advantages compared to on-board processing. First, simplifying DNNs may reduce model performance, and many state-of-the-art models are still computational intensive even after applying simplification techniques, *e.g.*, NasNet [164]. However, lightweight models that are suitable for mobile platforms may be far less accurate than the complex models, which can not provide satisfiable application performance. Second, it needs additional training and evaluating efforts to tune a simplified model. Offloading DNN tasks to servers can utilize server-class GPUs that achieve both low delay and high performance. Among mobile offloading systems for DNN workloads, MCDNN [60] studied the problem of scheduling variants of DNNs on the mobile and cloud for performance-resource trade-off. Neurosurgeon [82] presents the techniques to partition DNN computation between the mobile device and server at the granularity of neural network layers.

2.4 Our Work

This dissertation presents a new multi-tenant mobile offloading framework for real-time computer vision applications. Compared with the previous works, this work contains the

following novelties:

- **APIs:** Previous works consider simple offloading tasks, such as an object recognition and detection tasks [56, 37, 60, 82]. We explore systems that support complex vision applications. We present the DeepQuery system providing APIs to program vision tasks as queries, and build an application as a DAG of queries. The system caches intermediate results between queries to reduce the data communication between client and server.
- **DNN Serving Systems:** The previous works consider serving DNN models in the cloud, with CNN as the primary vision workloads. In this work, we focus on a different scenario: mobile vision applications backed by local GPU servers (the cloudlet) using a variety of DNNs. We propose solutions to several new challenges that are untapped. First, to support real-world mobile applications, we investigate a wide range of DNN models and their optimization techniques. Second, because GPU resources are limited on local servers, we design a serving system that co-locates models on shared GPUs to increase the resource utilization level. We also study adaptive batching method for DNN queries that consider both the performance requirements on latency and accuracy.
- **Low-latency Offloading Systems:** We present scheduling methods that addresses the multi-tenancy problem for CPU workloads in the cloudlet. It predicts the future offloaded tasks from all clients, detects potential resource contention, and coordinates tasks to reduce the contention. The system is able to maintain low processing latency even when the system load is very high (around 80% in our evaluations) at the cost of additional jitters of offloading intervals.

In the rest of this dissertation, we present the technical details of the new mobile offloading framework.

Chapter 3

DeepQuery: A Mobile Offloading Framework for Vision Applications

Emerging mobile applications can use several different vision algorithms and DNN models to obtain the capability to understand complex video content, such as cognitive assistant [56] and intelligent personal assistant [64]. To efficiently run these applications, a mobile offloading framework should be able to orchestrate multiple algorithms and to improve performance in aspect of delay and throughput, while considering multi-tenancy. In this Chapter, we present DeepQuery, a new mobile offloading framework crafted towards these goals.

3.1 Introduction

Advances in computer vision algorithms have fueled an explosion of mobile applications that extract contextual information from video streams [56, 64, 162, 160, 100]. Among these algorithms, DNNs are becoming a significant class of vision workloads on mobile platforms because of their superior performance in many challenging tasks, *e.g.*, object recognition, detection, and tracking [66, 164, 132, 65]. Going further, DNNs have been employed for vision tasks at higher semantic levels, such as detecting object relationships [155] and describing videos [156]. Therefore, we envision that emerging mobile applications can leverage multiple DNN models to extract rich and diverse information accurately from live video feeds, such as the video analytics application described in Figure 1.1.

To implement such applications using the offloading approach, we designed *DeepQuery*, a multi-tenant and real-time mobile offloading framework. It provides APIs to program a mobile application as a DAG of tasks to offload. On the server side, it manages a cluster of machines and processes offloaded tasks on CPUs or GPUs. Computer vision algorithms and DNN models are deployed as microservices. The following new features are implemented into DeepQuery:

First, DeepQuery provides APIs to cache mobile data in the form of key-value pairs on servers. An application is implemented as a DAG of queries that process mobile data, where each query corresponds to a vision task. With data management on the server side, intermediate or stateful data can be cached on servers to avoid redundant data transfer. For example, in Figure 1.1, object tracking and scene graph detecting tasks take the output of object detection task as their input. Using the data dependency information contained in the DAG, the system automatically caches the detection output on servers and forwards it to the downstream queries. Moreover, the APIs let applications share common processing modules and thus avoid redundant computation.

Second, the underlying serving system that processes client queries is optimized to attain low delays and high throughputs. We optimize DNN inference for a variety of vision tasks, *e.g.*, object detection, tracking, scene graph detection, and video description. For example, the FRCNN model [132] can be accelerated by classifying object proposals on multiple GPUs in parallel. Our experiments show that FRCNN (Inception ResNet V2) are $\times 1.7$ faster when being run on 4 GPUs. The speed-up is $\times 2.2$ for the more expensive FRCNN (NasNet) model. For the multi-object tracker using Siamese CNNs [32], DeepQuery batches multiple objects as the model input to improve the inference throughput.

In addition, we implemented scheduling algorithms into DeepQuery to reduce E2E delays when resource contention arises, for CPU and GPU workloads, respectively. We present the scheduling algorithms in Chapter 4 and 5.

3.2 Background

Video Analytics Systems: Video analytics systems that process a massive amount of queries on live video feeds in the cloud have been widely adopted in many application scenarios. For example, analyzing videos from surveillance camera networks installed in cities, organizations [100, 160, 162, 72], and smart hospitals [62]. Optasia [100] provides a set of SQL style APIs to build vision analytics applications. The queries are processed by user-defined operators in the SCOPE dataflow engine [35], comprising of design patterns such as extractors, processors, reducers, and combiners. It further optimizes the queries to reduce the redundancy of computation. VideoStorm [160] programs an application as a DAG of transforms on data, which are executed by processing modules. The placement of processing modules and the configuration knobs of transforms are decided dynamically at runtime.

Streaming massive live video feeds to the cloud may cause bandwidth shortage. To address this issue, Vigil [162] applies local processing to extract a fraction of valuable frames to be offloaded, according to specific user queries, to reduce the bandwidth usage. Using computing resources in local cluster at the network edge has several advantages. It avoids sending massive raw video to the cloud, and also provides low E2E delays. VideoEdge [72] considers a hierarchy of resources with diverse computing and network capabilities, *e.g.*, camera devices, private clusters, and public clusters. Query optimization techniques are proposed to make trade-off between resources and accuracy.

These video analytics systems, however, are not designed for the mobile offloading scenario, and therefore many important issues have not been addressed, *e.g.*, supporting RT tasks that require sub-second delays and techniques to enhance resource utilization. In addition, existing systems consider stationary cameras for surveillance purposes as the primary use scenarion. Compared to mobile cameras, video analysis can be simplified for many tasks. For example, detection and tracking of moving objects can use background subtraction methods [47]. As a result, such lightweight methods fall short for analyzing videos from mobile cameras.

Supporting and optimizing DNN workloads running on GPUs that are essential to mobile scenarios have not been addressed well.

Serverless Computing: Cloudpath [110] proposes a new serving methodology that is built upon a serverless cloud container framework and a distributed eventual consistent storage system implemented on Cassandra [89]. It dynamically installs lightweight event handlers that process mobile data and replicate data on demand in hierarchical clusters. LEVEA [158] adopts the serverless architecture on an edge computing platform consisting of local servers. It studies optimization methods to minimize the response time. However, these works consider only workloads running on processors, *e.g.*, face detection using the OpenCV library [34] and automatic license plate recognition implemented in OpenALPR [15]. However, these works have not considered DNN inference workloads. The service initialization phase may take much longer to load the model weights, which introduces new challenges in designing the optimization methods.

DNN Serving Systems: A DNN serving system manages trained models and processes tasks using the models. It enables Artificial Intelligence (AI) powered interactive services in the cloud and has attracted intense research interests [63, 40, 31, 3, 4, 6].

DjiNN [63] explores the idea of DNN-as-a-Service by designing centralized DNN service infrastructure. It supports the models for image, speech, and natural language processing (NLP) applications. The system targets at warehouse-scale servers (WSCs) with high processing throughput as the primary design goal, which is achieved using the batching and NVIDIA Multi-Process Services (MPS) [14] methods. Clipper [40] is a low-latency serving system that consists of two layers. It uses a model abstraction layer that exposes a common set of APIs as a wrapper of the underlying heterogeneous ML frameworks. The model selection layer implements methods to select and combine models dynamically to boost prediction accuracy. In pursuit of low latency and high throughput, Clipper adopts caching, adaptive batching, and straggler mitigation techniques. The open-source Tensorflow Serving framework [31, 12] provides a solution for high-performance production model serving. It supports model isolation for multi-

tenancy purpose. The threads that process DNN queries are separated from long operations such as model loading. The framework provides easy-to-use APIs for multi-model serving. Dynamic batching and latency control capabilities according to user SLAs are also included.

Queries using DNNs require over $100\times$ computational resources over traditional web queries [64]. As the computational complexity is growing dramatically, hardware accelerators for executing DNNs, such as FPGAs and ASICs, becomes commonplace in cloud clusters. To explore the options of optimizing latency and throughput, as well as cost, Sirius [64] conducts evaluations on platforms of processors, GPUs, manycore co-processors, and FPGAs, using benchmarks of IPA applications including voice and image queries. It comprehensively discusses the acceleration performance on GPUs and FPGAs. Because batching inevitably diminishes latency performance, Project Brainwave Neural Processing Units (BP NPUs) [51] leverages computational parallelism within individual DNN inference queries to improve throughput. The system adopts a single threaded Single Instruction Multiple Data (SIMD) architecture and a pipeline parallelism method to achieve high hardware utilization without batching. Evaluations on FPGAs demonstrate that it achieves both low latency and high throughput in processing DNN queries.

Most previous works serving DNN in the cloud consider Convolutional Neural Networks (CNNs) as the primary vision workload. This is in contrast to the scenario envisioned here: mobile vision applications piggybacked on local GPU servers (i.e., cloudlet) using a variety of DNN models. To be more specific, we extend previous works by investigating a richer set of DNN models, and co-locating DNN inference tasks with different timing requirements on shared GPUs for a higher utilization (Chapter 3.4).

3.3 Applications

We use the following applications to motivate and evaluate our work. The applications include conventional computer vision algorithms running on CPUs and DNNs running on GPUs.

We consider two types of applications in the aspect of timing: real-time (RT) and non-real-time (non-RT) applications.

3.3.1 CPU Workloads

Face Detection: *FaceDetect* detects human faces continuously from streaming video data. We use the OpenCV [34] implementation that adopts Haar feature-based cascade classifiers [149]. The algorithm applies feature filters on small windows of the image and uses the weighted sum to produce the result. Instead of fully processing all windows, a window is discarded as soon as it fails any stage of filters, therefore the total computing time highly depends on the image and its variability across frames can be large. Our implementation is single-threaded.

Stereo Feature Matching: *FeatureMatch* detects interest points in a video stream captured from a binocular camera (a pair of left and right cameras), extracts features (e.g. SIFT [98], SURF [30]), and matches features. It outputs pairs of matched points which are to be used for generating a depth map. Our implementation is based on OpenCV routines. It computes SURF features of the left and right frames in parallel on two threads. It then runs brute-force matching on one thread. Because the number of interest points depends on the content of a frame, the computing time can be highly variable.

Neural Network Object Detection: *ObjectDetect* detects a set of objects in a video stream. It localizes objects and labels each with a likeliness score. We use the open source real-time object detection system “You only look once” (YOLO) [131]. The implementation is finely parallelized and uses all cores in a typical processor in parallel. It has a constant computing time on each video frame.

Timely response is a significant requirement for the usability of these RT applications. For example, *FaceDetect* and *ObjectDetect* lose trackability as delay increases. *FeatureMatch* can be used in robotics and autonomous systems to retrieve depth information from a binocular camera. For all applications above, it is important to attain low E2E delays despite offloading. The three applications present differences in computing time variability and parallelism. We use

their differences to explore the design for a general and highly usable offloading framework.

3.3.2 GPU Workloads

We describe the vision tasks and the DNN models used in the example application (Figure 1.1), including object detection, tracking, scene graph detection, and high level vision tasks such as scene graph detection and video description. We also characterize the runtime overhead of the DNN models to illustrate the challenges that drive the design of DNN model serving systems.

3.3.3 Platform Specifications

The experiments that characterize the DNN models are conducted on servers running Ubuntu 16.04 with a 48-core Intel Xeon Gold 6136 processor with 93GB system memory, installed with CUDA 9.0 and cuDNN 7.1. We use up to 4 NVIDIA GeForce GTX 1080 Ti GPUs (11GB memory and 11.3TFLOPs each device) on one server. The DNN models are implemented using Tensorflow 1.5.1 [24].

3.3.4 CNN Feature Extraction

A CNN extracts a fixed length vector representation from an image, which can be used in a wide variety of vision tasks. A feature extractor has the same architecture with a CNN based object classifier [70, 66, 164]. The difference is that a feature extractor removes the last fully-connected classification layer, and outputs a feature vector instead of a classification label. Batching multiple queries in DNN inference is effective in improving throughput [63, 40]. The throughput gain depends on the network structure and size of the model. We measure inference delay for two popular CNNs, VGG-16 [138] (130 million parameters) and ResNet-152 [66] (60 million parameters), as shown in Table 3.1. The *SD* field in the table gives the standard deviation. The throughput of ResNet-152 is improved by $\times 2.3$ by using a batch size of 16. However, VGG-16 does not benefit from batching because the GPU already saturate without batching.

This work considers the feature extraction task as delay-tolerant (non-RT) workloads. For example, extracting CNN features of frames in a video stream to be used by video description tasks. We investigate scheduling techniques to co-locate them with RT tasks, and reduce their influence on RT tasks.

Table 3.1. DNN Inference Delays of CNN models (SD) (ms).

Images	1	2	4	8	16
ResNet-152	14 (1)	19 (1)	30 (2)	56 (2)	99 (2)
VGG-16	56 (1)	114 (3)	206 (4)	488 (5)	-

3.3.5 Object Detection in Videos

Video object detection (VID) is the essential building block in many continuous vision applications, such as cognitive assistant [56]. It detects and tracks multiple objects online in a live video stream. VID generates traces of tracked object in continuous frames, *i.e.*, tubelets. An object is represented by a bounding box with a class label and an unique object identifier. We adopt tracking-by-detection [61], a generic VID scheme. It detects objects in static video frames, and then associates the detected object boxes into sequence to obtain tubelets. To reduce the high computation overhead of object detections, the more lightweight object tracking is used to track objects across frames. Therefore object detection can run at a lower frequency. Hungarian algorithm [87] is used to associate newly detected objects with currently tracked objects. It uses the intersection over union (IoU) of two object boxes as the association metric.

Two types of task, noted as TRK and DET, are generated by the VID application, as shown in Figure 3.1. A DET task consists of four sub-tasks: tracking objects from the previous frame, detecting objects in the current frame, fusing the results, and initializing the object tracker. The period is P_{TRK} for TRK and P_{DET} for DET. Figure 3.1 illustrates the GPU loads that VID tasks generate as well. For a DET task, the tracking and detection sub-task can run in parallel on multiple GPUs, or multiple servers. If DET_i takes more than P_{TRK} to complete, TRK_{i+1} will be delayed due to input data dependency, where i is the task index.

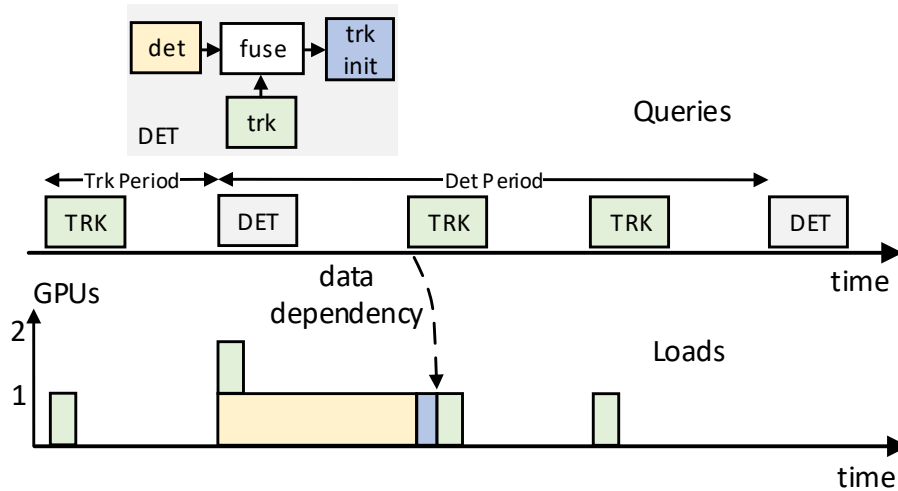


Figure 3.1. A video object detection application composed of TRK and DET tasks: A detection task contains sub-tasks including detecting objects in the new frame, tracking objects from the previous frame, fusing objects, and initializing the tracker. DeepQuery programs the sub-tasks as queries and uses predicted future GPU load in the scheduling algorithms, in order to maximize server utilization.

Object Detection: We use the Faster Region-based CNN [132] (FRCNN) model to detect objects. The structure of a FRCNN model is shown in Figure 3.2. There are three stages: a Regional Proposal Network (RPN) stage detecting a fixed number of object proposals, a classification (CLS) stage classifying proposals, and a post-processing (POST) stage running the Non-Maximum Suppressing (NMS) algorithm and converting classified proposals to object bounding boxes.

The detection performance and speed of FRCNN detectors are decided by two configuration parameters: the convolutional layers and the number of object proposals. Here we compare the delays of three detectors provided in the Tensorflow Detection Model Zoo [17]: Inception V2 (100 proposals), Inception ResNet V2 (300 proposals) and NasNet (300 proposals). Their detection performance (mAP) on the COCO dataset [93] is 28%, 37% and 43% respectively [17]. As shown in Table 3.2, we observe that the throughput of a lightweight model, Inception V2,

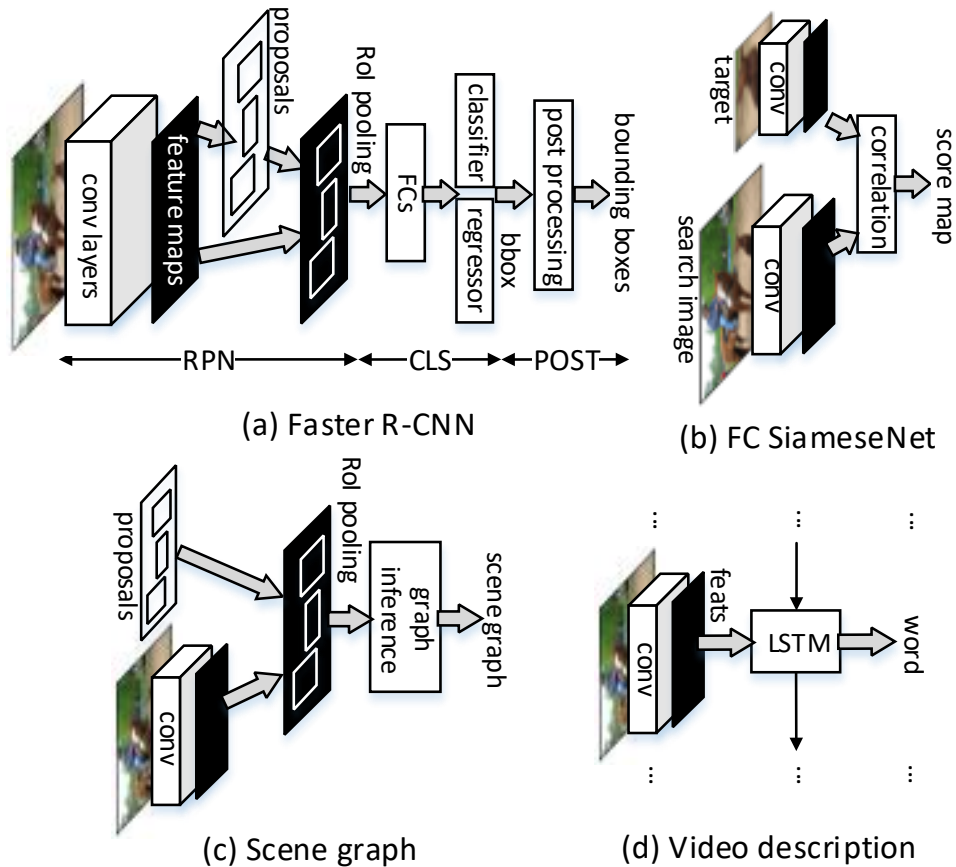


Figure 3.2. DNN models for (a) object detection (Faster R-CNN [132]), (b) object tracking (FC SiameseNet [32]), (c) scene graph detection [155] and (d) video description [148].

can be improved by $\times 1.3$ using a batch size over 8. However, deeper models such as Inception ResNet V2 and NasNet does not benefit from batching on NVIDIA GTX 1080 Ti GPUs.

Observed in Table 3.2 and the detection performance given by [17], heavyweight models with high detection performance are usually slower, with delays from hundreds of milliseconds to seconds. It limits their use in real-time applications. To speed up these object detectors, we present accelerating techniques that parallelize the classification stage on multiple GPUs in Section 3.4.

Object Tracking: We use a high performance multi-object tracker based on the Fully-

Table 3.2. DNN inference delays (ms) of object detection models (with the standard deviation in parenthesis).

Images	1	2	4	8	16
InceptionV2	38 (2)	64 (2)	125 (2)	239 (4)	478 (8)
Inception-ResNet V2	370 (4)	728 (5)	1415 (8)	2863 (11)	-
NasNet	1025 (8)	2063 (8)	-	-	-

Convolutional Siamese Networks (FC Siamese) [32]. The network structure is shown in Figure 3.2. The tracker extracts CNN features from both the target and the search region, and then localizes the target using a score map obtained from cross-correlation of the features. Accordingly, the tracking process contains two stages. The initialization (INIT) stage extracts features of a target, and the tracking (TRK) stage extracts features of search regions in a new frame and localizes the target. To track multiple objects, the model takes multiple objects as the input of TRK, which has the form $(im, [trk_states...])$, where im is a video frame, trk_state is a tracking state including target features and location. The input is transformed to $([search_regions...], [target_regions...])$ as an input batch for the Siamese Network. The outputs are bounding boxes of tracked objects, and tracking states with updated target features and location.

Our implementation tracks multiple targets in a single DNN inference. The delay of a single tracking task is thus decided by the number of targets. Table 3.3 summarizes the amount of delays for processing different numbers of targets as a batch. Using a batch over 8 increases the throughputs by $\times 1.3$ for INIT and TRK, compared with processing a single object. To improve overall throughput, for tracking tasks which have few targets, several tasks can be batched as $([im_i...], [[trk_states...]]_i...)$. The batching should be aware of timing requirements for delay-critical tasks, because it increases the processing delays.

Table 3.3. DNN inference delays (ms) of object tracking models (SD).

Objects	1	2	4	8	16
Trk-Init	6 (0)	10 (0)	18 (0)	37 (1)	74 (2)
Trk-Trk	19 (1)	32 (1)	60 (2)	115 (2)	230 (3)

3.3.6 High-Level Vision Tasks

High-level vision tasks extract contextual information at higher semantic levels. In this work, we consider them as non-RT workloads and investigate scheduling techniques to co-locate them with RT tasks, as well as to reduce their influence on RT tasks.

Scene Graph Detection: A scene graph detection task (SG) takes object proposals in an image as input, detects objects and infers the relationships between objects. We consider the model proposed in [155] that has three stages: generating object proposals, extracting VGG-16 image features, and inferring objects as well as their relationships using iterative message passing. Different from object detection, object class labels are inferred using both the features of objects and object relationships.

We measure SG delay excluding the object proposal stage, as given in Table 3.4. The delay includes a VGG-16 delay around 56ms (Table 3.1). Similar to multi-object tracking, the delay depends on image context, *i.e.*, number of object proposals in this case. Scene graph models usually select a small set of high quality proposals, because most of proposals are non-object and it is wasting to process them in the graph inference. In our implementation, the proposals are generated from an object detector using a low confidence score threshold (0.01).

Table 3.4. DNN Inference Delays (ms) of scene graph models (SD).

Object Proposals	2	4	16	64
Scene Graph	62 (1)	59 (2)	71 (1)	249 (3)

Video Description: A video description task generates a sentence in natural language from a video clip. We adopt the sequence to sequence model [148]. It takes a sequence of CNN features of video frames as the input, which is encoded by a stacked LSTM with two hidden layers [69]. The LSTM decodes it into a sequence of words. Our implementation takes video clips of 80 frames as the input. The CNN features have a dimension 4096. The serving system can improve the throughput of video description tasks by batching video clips. As shown in Table 3.5, the throughput increases by $\times 20.9$ with a batch size of 256.

Table 3.5. DNN Inference Delays (ms) of video description models (SD).

Videos	1	16	64	128	256
S2T	48 (4)	52 (2)	206 (14)	323 (14)	589 (24)

3.4 Optimizing DNN Inferences

We discuss the optimization techniques to improve processing delays and throughputs for the models, from the perspective of DNN serving systems.

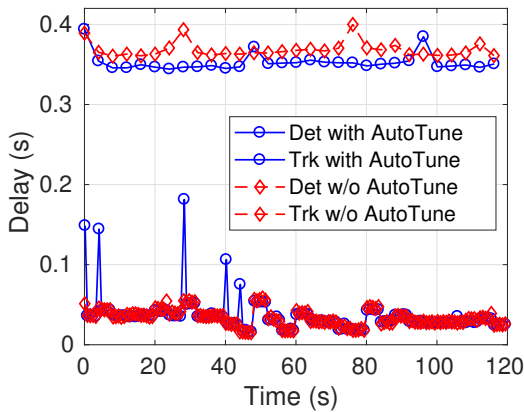
3.4.1 Customized Input Batching

As discussed in the previous section, DNNs may take visual data in different forms as input. The DNN serving system should support customized methods to batch input data for specific models. Therefore the concept of a batch becomes broader than images, *e.g.*, it can refer to objects or video clips. This work implements customized batching methods for the models.

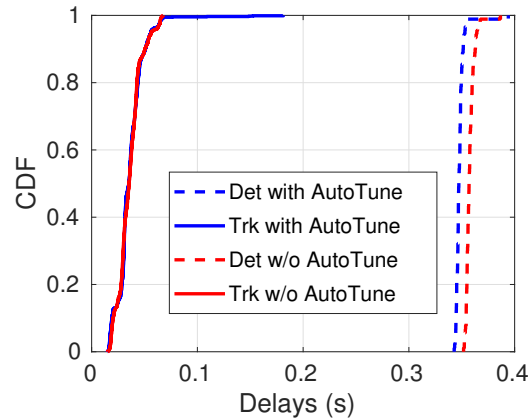
3.4.2 Sharing GPUs by Multiple Models

Conventional DNN serving systems deploy one model on a GPU device. It uses dynamic batching [40, 63], as well as multitasking on GPUs [14, 63, 151, 119], to improve resource utilization and thus processing throughput. The situations that we consider in this work is different. We focus on RT applications with delay constraints at sub-second scales, therefore we have a GPU device run one model at a time, to minimize delay for RT tasks. Batching is used for non-RT tasks, and for RT tasks when multiple tasks are waiting for processing. On the other hand, on the cloudlet servers, because GPU resources are much less than the cloud, we consider deploying multiple models on shared GPUs. A GPU can thus process queries using different DNN models for higher resource utilization levels.

Interference: Several DNN models can run on shared GPUs to improve resource utilization on cloudlet servers. Interference caused by co-locating DNNs is an important performance issue to be considered. Intuitively, inferences do not exist when different DNNs run in sequence



(a) DNN processing delays.



(b) CDFs of delays.

Figure 3.3. DNN processing delays for object tracking and detection with and without cuDNN AutoTune. When multiple DNN models run on a shared GPU, AutoTune may incur delay spikes when it profiles input data for a model. We switch off AutoTune to avoid delay spikes and improve the accuracy of delay prediction.

on a GPU. However, as shown in Figure 3.3a, delay spikes of tracking tasks (the blue lines) are observed, when the tracking and detection models are on the same GPU. The *AutoTune* function that profiles input data to optimize batching processing in the *CuDNN* library is the source of delay spikes. Re-profiling happens when DNN workloads change and causes delay spikes. Figure 3.3a shows that the spikes are eliminated after turning off AutoTune. The advantages and drawbacks of AutoTune are clearly illustrated in Figure 3.3b via the Cumulative Distribution Functions (CDFs) of delays. AutoTune speeds up FRCNN by around 10ms, whereas it results in long tails of TRK delays. In the system, we tune off AutoTune for machines that co-locating models for RT tasks and other workloads, to avoid unpredictable delay spikes.

Preempting GPU Workloads: The resource utilization can be improved by co-located RT and non-RT tasks on shared GPUs. It requires that non-RT tasks yield GPUs when any RT tasks are ready to run. On CPU processors, it is enabled by *preemption*. However, without special handling in context switches enabled by driver extensions, preemptions incur large overheads in both processing delays and throughputs for GPU workloads [142, 118]. However, such extensions are not yet provided on the latest GPUs. This work solves the preemption problem

by predicting future RT tasks and accordingly limiting the budget to run non-RT tasks, to end processing before RT tasks starting (Section 4.4.2).

3.4.3 Parallelization

For object detection, although many fast lightweight model have been propose [70, 74], heavyweight DNNs [164] still demonstrate much superior performance [17]. The parallelization technique is able to execute DNNs with both high performance and low delay, which is valuable for RT application. A FRCNN model can be expedited by processing the CLS stage on multiple GPUs in parallel. Each GPU classifies a portion of the proposals generated by the RPN stage. We test the delay improvement by evenly allocating the proposals on 1, 2 and 4 GPUs. As given in Table 3.6, using parallelization on 4 GPUs, FRCNN (Inception ResNet V2) is $\times 1.6$ faster, and the more compute intensive FRCNN (NasNet) model is $\times 2.2$ faster.

Table 3.6. Precessing Delays (ms) of FRCNN stages using different numbers of GPUs: Data parallelization can effectively accelerate DNN inference.

ResNet Inception V2				
#GPU	RPN	CLS	POST	Total
1	144 (3)	245 (1)	13 (2)	382 (3)
2	140 (5)	119 (1)	13 (2)	274 (7)
4	140 (3)	63 (3)	13 (3)	219 (8)
NasNet				
#GPU	RPN	CLS	POST	Total
1	244 (4)	747 (4)	13 (3)	1004 (8)
2	243 (3)	382 (2)	10 (1)	637 (3)
4	243 (3)	194 (1)	11 (1)	448 (3)

Similar processing pipelines as FRCNN, which extract regional features from object proposals, have been adopted by a wide variety of models, *e.g.*, bottom-up image description tasks using object features [29]. The infrastructure support that speeds up FRCNN is thus useful for these models as well. We describe how parallelization is implemented in Section 3.6.1.

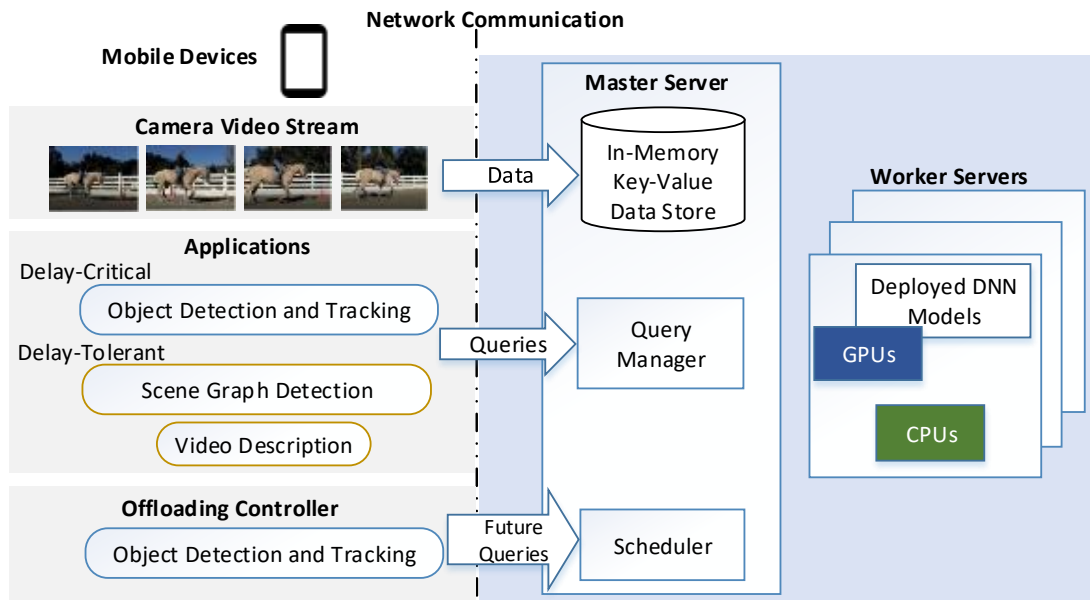


Figure 3.4. An overview of the DeepQuery system: The master server contains an in-memory database to cache data, a query manager to manage queries that process data, and a scheduler to coordinate query executions. Queries are executed on the worker servers equipped with GPUs.

3.5 Programming Applications in DeepQuery

DeepQuery is a serving system for vision applications on mobile platforms. Figure 3.4 illustrates the architecture of DeepQuery: Mobile clients offload vision applications using DeepQuery’s APIs to upload data and submit queries on the data. On the server side, the *master* server manages data in a distributed in-memory key-value database. When the input data of a query are complete, the master dispatches it to *worker* servers for processing, which deploy applications as micro-services. The *scheduler* selects workers to dispatch queries, and controls batch sizes of DNN inputs.

3.5.1 Programming Model

A **query** defines a vision workload to offload. It processes data using one or more DNN models, or other vision algorithms. An application can submit multiple queries to the server. The

server creates a corresponding **job** for each query, which contains a DAG of processing **stages**. A stage is the atomic unit of data processing, corresponding to a DNN model or an algorithm served as a micro-service. For example, FRCNN can be spitted to a pipeline of RPN, CLS and POST stages, where we can parallelize the CLS stage. The input and output data of stages are cached in memory (not the key-value database). A stage becomes runnable when its predecessor stages finish and all input data are ready. As long as a job has runnable stages, it stays in the ready queue and can be fetched by job runners. Therefore it can run in parallel on multiple GPUs.

3.5.2 Application Programming Interfaces (APIs)

DeepQuery APIs separate operations that manage data and queries that process data. There are three types of operations:

Data Operations: Data operations manipulate data stored on the server, including imagery data from mobile camera and processing results of queries. All data are stored in a distributed in-memory database as immutable key-value pairs. The basic operations are ADD, GET, DEL to write, read and delete a keyed data. We use Redis [16], a widely used in-memory data structure store, as the underlying implementation of our database. The key contains a name field and an index field. For example, a client adds the i th frame to the database on server with “im:i” as the key, with “im” as the name, and i as the index.

The data store supports higher level stream and batch operations that wraps the basic operations to handle a collection of key-value data. For example, a video stream that contains frames with indexes from i to $j-1$ can be represented as “im:[i:j]”. Besides image, object bounding boxes of an image is a frequently used data element, with a name like “boxes:i”.

Queries: Basically, the query specifies the type of the algorithm or DNN model, the input and output data keys. The system routes the query to a worker server that serves this algorithm or model. After running the query, the output data are paired with the output keys. They are cached on server or sent back to the client, controlled by data operations. A query may

contain additional information such as parameters of vision algorithms.

FutureQueries: The scheduler of DeepQuery needs the information of future queries to predict GPU loads (Section 4.3). The prediction is based on measured DNN processing delays, network delays and data dependencies between queries. It generates GPU loads as in Figure 3.1. The application client may adjust offloading configurations. For example, the VID application may adjust the tracking and offloading periods, or DNN model types, to tune performance and computational overhead. The FutureQueries APIs are used to keep the scheduler updated with the future queries. When a streaming RT application sends the i th offloading request, it needs to attach the information about the next $(i + 1)$ request. The information includes starting times (t_{start}), DNN model types (m) and input batch sizes (b) of queries in the next request.

Sample Offloading Request: Listing 3.1 gives the example of an offloading request for DET task in the VID application. The task contains four queries: tracking objects in a new frame (“im:i”) using the tracking states (“trk_states:i-1”), detecting objects, and fusing the tracking (“t_boxes:i”) and detecting (“d_boxes:i”) results. At the end, the tracking states are initialized for the fused boxes. In the future loads field, the next query is a TRK task, with a batch size estimated as the number of currently tracked objects.

For each query, the microservice and method fields specify the microservice that processes this query by invoking a given method. We provide the details of microservice in Appendix A.

3.5.3 Data Management

The data are stored in a distributed in-memory key-value database on the master server. The data types must have methods of serialization and de-serialization. Therefore it can be transferred among the client and the servers. A data element is stored in one of three forms: a Python object, serialized data in memory, and serialized data on disk. In default, a data element is cached as a Python object, and it is dumped into disk after a timeout to save memory usage. The client can specify customized rules to persist data through data operation APIs.

A worker server has a slave database with a partial replication of the data. It is synchro-

Listing 3.1. An offloading request for object detection (DET) tasks in video object detection.

```
1 Request {
2   dataOps=[
3     DataOp(op=ADD, k="im:i", v=image),
4     DataOp(op=GET, k="boxes:i") ],
5   queries=[
6     Query(microservice=OBJECT_TRK, method=TRK,
7       inputs=["im:i", "trk_states:i-1"],
8       outputs=["t_boxes:i", "trk_states:i"]),
9     Query(microservice=OBJECT_DET,
10      inputs=["im:i"],
11      outputs=["d_boxes:i"]),
12     Query(microservice=FUSE_OBJECTS,
13      inputs=["t_boxes:i", "d_boxes:i"],
14      outputs=["boxes:i"]),
15     Query(microservice=TRK, method=INIT,
16      inputs=["im:i", "boxes:i"],
17      outputs=["trk_states:i"]) ],
18   futureQueries=[
19     FutureQuery(microservice=TRK, method=TRK,
20      time=t_now+period, batchSize=nr_boxes)]
21 }
```

nized using a master-slave strongly consistent protocol. When the master server dispatches a query to run on a worker, it checks whether the input data exist in the worker database. If not, the missing data are transferred to the worker. This design speeds up applications with states, *e.g.*, object tracking has target features and locations as its states. The states can be pre-cached at the worker to remove the data transfer from the critical path of query processing. Deciding which data are pre-cached at each worker is a trade-off between memory usage and computing efficiency. The strategy to cache states on workers are described in Section 5.6.3.

3.6 Processing Queries on Worker Servers

In this section, we describe the design of workers that process queries, for workers equipped with GPU devices that process DNN queries, and workers that process queries on CPUs, respectively.

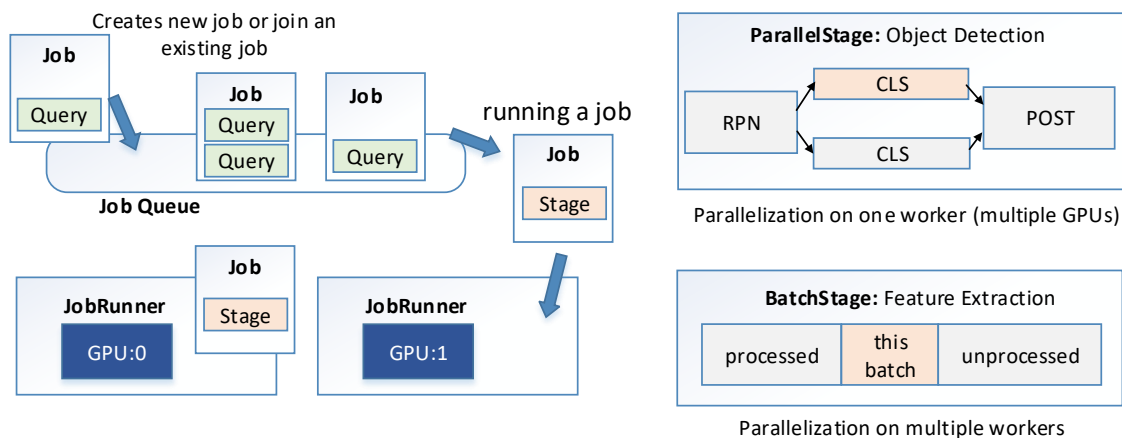


Figure 3.5. Query processing on the worker servers. A job is created for each query. A job runner manages job executions and loaded DNN models on a GPU. A ParallelStage can run on multiple GPUs to accelerate DNN inference using data parallelization. A BatchStage processes a batch of input data on multiple worker servers.

3.6.1 Processing DNN using GPUs

A worker server manages one or more GPUs. A GPU is wrapped by the *JobRunner* module, which exposes APIs to deploy or remove DNN based micro-services, and process queries. To use a DNN model in DeepQuery, the micro-service interfaces must be implemented, including methods to initialize the model, batch multiple queries, and process input data. A GPU can serve multiple DNN models. In the current implementation, we manually specify the models that can be co-located on the same device. Automatic placement algorithms lie in our future works as discussed in Chapter 8.

The worker creates job to process queries. As shown in Figure 3.5, runnable jobs are buffered in a ready job queue. Each job runner fetches jobs to run from it. In this work, a job runner executes one job at a time, which means the GPU device is exclusively used by a DNN model. RT jobs are prioritized in the ready queue. Jobs with the same timing requirement are ordered by the query starting timestamp t^{start} .

Parallel Stages: A *parallel* stage runs a DNN inference on multiple GPUs on one

worker through data parallelization. It can speed up parallel sessions of DNN models, such as the CLS stage of FRCNN. The implementation adopts the *fork-join* execution model. It requires customized methods to split the input data and combine the output data. For FRCNN with 300 proposals, the outputs of the RPN stage include the coordinates of proposal boxes (shape 300×4 , serialized size 4.8KB), and the extracted CNN features (shape $75 \times 100 \times 1088$, serialized size 32MB). A job runner can run a part of CLS by taking a sub-batch of proposal boxes, then combines the classifications obtained on all runners to get the final result. The algorithm to dynamically decide the batch size to run on each GPU is given in Section 4.5.1. We do not consider processing a FRCNN inference task in parallel on multiple workers, because of the large size of CNN features. It results in large overheads in serializing, transferring and de-serializing the data.

Adaptive Batching: The worker automatically batches queries of lightweight DNN models in the ready job queue. Before creating a job for a query, the worker first check whether there are already this type of jobs in the ready queue. If so, the query is appended to an existing job. Considering delay increase due to batching, the system sets a default limit of the per job batch size for each DNN model. No more query can be added if the batch size of the job exceeds the threshold. Because compute intense models benefit less from batching, the thresholds are low for such models. A query can also specify a customized batch size limit, depending on its timing requirement.

3.6.2 Processing Queries on CPUs

To process queries using CPU processors, a worker machine hosts one or more *computing engines*. Each computing engine serves one offloading application encapsulated in container. Each engine processes one task at a time and does not allow concurrent tasks. Our implementation adopts *Docker* [5] containers. We use Docker’s resource APIs which currently support specifying processor share, limits and affinity, as well as memory limits for each container. The total number of processor cores on worker W_k is denoted as W_cpu_k .

A worker can have multiple engines for the same application in order to fully exploit multi-processors. For example, to serve a single thread application on a dual-core machine, two engines should be hosted. A worker can also host different types of engines to share the machine by various applications, to gain resilience against usage change of applications. As a result, the total workload of all engines on a worker may exceed the limit of processor resource (W_{cpu}). We classify workers into *reserved* and *shared* which adopt different scheduling strategies:

- **Reserved:** On a reserved worker, the sum of processor usages of engines never exceed W_{cpu} . Therefore whenever there is a free computing engine on reserved worker, it is guaranteed that dispatching a task to it does not induce any processor contention. Naive First-Come-First-Serve (FCFS) scheduling works for reserved workers.
- **Shared:** In contrast, the total workload on shared worker may exceed W_{cpu} . As example in Figure 3.6, a dual-core machine hosts two FaceDetect engines ($T_{paral} = 1$) and one FeatureMatch engine ($T_{paral} = 2$), where T_{paral} is the number of CPU cores used by a task. With this deployment both applications are able to fully exploit dual-cores. When there is a running FaceDetect task, an incoming FeatureMatch task will cause processor contention. *Load-aware scheduling* is used for shared workers which is described in Section 5.4.3.

The techniques that remove resource contention using the Plan-Schedule approach and schedule tasks to workers are presented in Chapter 5.

3.7 Summary

This chapter presents DeepQuery, a mobile offloading framework for computer vision applications offloaded to remote CPUs and GPUs. DeepQuery provides a set of APIs to manage mobile data on servers as in-memory key-value pairs, and to program vision tasks as queries that process mobile data. We present the system design to deploy DNN models running on GPUs,

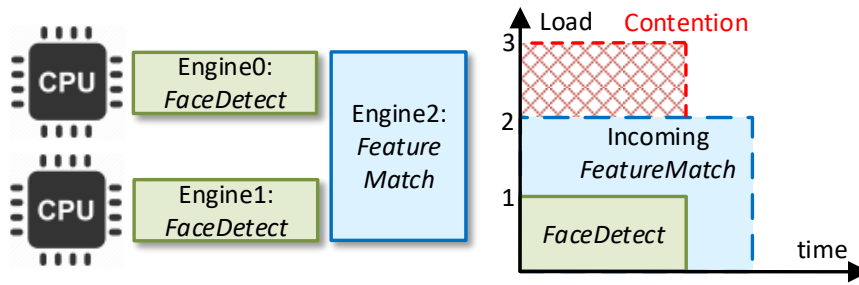


Figure 3.6. A dual-core worker machine have two engines of FaceDetect ($T_{\text{paral}} = 1$) and one engine of FeatureMatch ($T_{\text{paral}} = 2$). The plot on right gives an example of processor contention on a shared worker between a running FaceDetect task and an incoming FeatureMatch task.

and other algorithms running on CPUs encapsulated in Linux containers. It conducts several optimizations in serving DNN models, *e.g.*, reducing inference delay via data parallelization on multiple GPUs and improving throughput via input batching.

This chapter, in part, is a reprint of the material as it appears in Zhou Fang, Mulong Luo, Tong Yu, Ole J. Mengshoel, Mani B. Srivastava, and Rajesh K. Gupta. “Mitigating Multi-Tenant Interference in Continuous Mobile Offloading”, in Proceedings of the 2018 International Conference on Cloud Computing (CLOUD’18), 2018. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Scheduling DNN Queries for High GPU Resource Utilization

Executing DNN queries from a group of clients in a cloudlet may have resource shortage issues. DNN queries of RT applications should run on under-utilized GPU servers to avoid contentions that increase delays. On the other hand, there are non-RT applications for which throughput is the primary performance consideration. In this chapter, we present scheduling algorithms to co-locate RT and non-RT applications on shared GPU devices for a high resource utilization, while providing low delays for RT applications by mitigation contention on GPU resources.

4.1 Introduction

Compared to conventional computing resources, *i.e.*, processors, server-class GPUs are more expensive resources. Use the AWS cloud computing platform [2] as example, for on-demand VM instances, the per-hour price of an instance with 4 vCPUs¹ with a NVIDIA GK210 GPU (p2.xlarge) is around 4 times of the price of an instance without GPU (c4.xlarge). Improving GPU utilization can therefore reduce the monetary cost of cloud services. For local computing resources (*e.g.*, a cloudlet), the amount of GPUs is limited and thus resource shortages may take place. In this case, performance degradation or denial of service may happen because

¹A vCPU is a hyperthread of an Intel Xeon CPU core.

there are no dynamically provisioned resource in a cloudlet. By observing these facts, methods to improve the resource utilization of GPUs are valuable for a mobile offloading and serving system.

Co-locating workloads with different timing requirements can improve resource utilization [92, 36]. In our scenarios, as described in Section 3.3, DNN queries generated from mobile vision applications can be classified into real-time (RT) tasks that desire low delays, *e.g.*, object detection and tracking, and non-RT tasks that desire high throughput, *e.g.*, video description. By co-locating RT and non-RT tasks on shared GPUs, the idle time slots when GPUs are not processing RT tasks can be utilized by non-RT tasks to boost resource utilization. The challenge is that resource contention between RT and non-RT tasks must be carefully mitigated to avoid delay degradation.

In this chapter, we present scheduling algorithms to co-locate RT and other non-RT tasks on shared GPUs that are implemented into the DeepQuery system, targeting at the cloudlet scenario. To avoid resource contention on RT tasks, non-RT tasks should yield GPU resources to any ready RT tasks. However, it is challenging for tasks running on GPUs due to the lack of support for preemptive multi-tasking. To overcome this issue, we designed a new *predictive* and *planning-ahead* method. It predicts future RT queries using the application information provided by the client, as well as estimated DNN processing and network delays. Dynamic batch size control is applied to non-RT DNN tasks to finish their processing windows before any future RT task using the resource.

In addition, the scheduler has other functionalities based on the prediction of GPU workloads, such as *dynamic batch size adaption* in data parallelization to accelerate a DNN model, and *worker selection* for incoming queries to balance GPU load on each server. We evaluated the scheduling methods of DeepQuery in a variety of deployments on servers equipped with NVIDIA GTX 1080 Ti GPUs, using several real-world applications.

4.2 Background

Scheduling DNN Tasks: DNN serving systems, *e.g.*, Clipper [40], Djinn [63], and Tensorflow Serving [31], adopt query batching to improve throughput. Djinn further studies using NVIDIA Multi-Process Service (MPS) [14] to run multi-process CUDA applications, typically MPI jobs, for higher GPU utilization. Clipper [40] selects the appropriate model type considering the requirements on model accuracy and processing delay. It also presents straggler mitigation techniques to reduce the tail delay. These works do not consider co-locating RT and non-RT tasks, which is addressed in our work.

Scheduling GPU Workloads: Multi-tasking techniques on GPUs have attracted intense research interest due to increasingly more workloads running on GPUs. To co-locate RT and non-RT tasks on GPUs, preemption is necessary to yield resource to urgent tasks. To have similar preemptive multi-tasking capabilities on GPUs, a set of hardware extensions that allow GPUs to efficiently support preemption of GPU workloads has been proposed in [142]. Chimera [118] designs streaming multi-processor flushing to reduce the overhead of context switch. These techniques, however, are still not available on modern GPUs, which are limited to be adopted in our work.

Co-locating RT and non-RT Tasks: Latency-critical cloud workloads, such as web queries, are usually located on separate machines from batching workloads such as offline data analysis. It guarantees low delay for latency-critical workloads, however, it leads to low utilization levels on server machines [92]. Workload co-location on CPUs has been studied in [92]. It schedules latency-critical workloads in an interference aware manner when batching workloads present. The original Linux Completely Fair Scheduler (CFS) is replaced by a new scheduler that addresses latency requirements and fairness of co-located workloads. This work explores workload co-location for tasks running on GPUs.

Baymax [36] explores the scheduling methods for co-locating user-facing and throughput oriented tasks on shared non-preemptive accelerators, using GPUs as the example. A user-

facing task comes with a QoS target that is a deadline of end time. The scheduler estimates task execution times, calculates the free GPU time that co-located applications can run (*QoS headroom*), reorders and delays ready tasks to avoid QoS violation. This work considers a similar problem that RT and non-RT DNN tasks are co-located, with a different design goal to reduce the resource contention between RT and non-RT tasks as much as possible.

4.3 Scheduling System Design

The scheduler for DNN workloads in DeepQuery has three functionalities:

- For parallelizable and batching DNN queries that can be processed on multiple GPUs in parallel, the scheduler decides the optimal batch size to run on each device.
- For co-located RT and non-RT queries, it prevents non-RT queries to run, or limits its batch size, to avoid GPU resource contention with future RT queries.
- It routes queries to workers based on the estimated GPU load on each worker.

All these functionalities are based on the capability to predict future GPU load and detect GPU resource contention.

4.3.1 Application Timing Characteristics

The system supports three timing types of DNN queries. **Streaming RT** queries are generated from delay-critical continuous mobile vision applications, which are the primary workloads in mobile offloading frameworks [56, 37]. The DeepQuery scheduler prioritizes their executions over other queries. **Non-RT** queries are delay-tolerant. **Batching** queries are non-RT, and they process batches of input data. Non-RT queries do not have tight delay constraints and are served by the system in a best-effort manner. Mobile applications may offload a mixed set of RT and non-RT queries. In the example application in Figure 6.1, VID submits streaming RT TRK and DET queries. SG queries are non-RT, and CNN queries in video description are batching.

4.3.2 Dispatching Queries to Workers

When the system is composed of multiple workers, the scheduler routes queries to appropriate workers based on the supported micro-service types and the current GPU load of each worker. It uses different strategies for different query types.

- **Stream RT Queries:** For a streaming RT query Q , the scheduler receives its information as a future query Q^{future} , attached with the previous offloading request. The scheduler selects the worker to dispatch Q at this time, ahead of when the real query arrives. It is necessary because Q^{future} must be allocated to one worker W_k , to update its set of future queries $\{Q^{future}\}_k$. The worker selection algorithms are based on predicted GPU contention and load. The details are given in Section 4.4.3.

The scheduler selects the worker W_k for Q before it arriving. For applications with states, or queries that take the output of previous queries as the input, the input data that are available before Q arriving can be sent to W_k and be pre-cached in the worker database. It avoids transferring the data when Q arrives to reduce query processing delay.

- **Non-RT Queries:** The worker to dispatch a non-RT query is selected when it arrives. Because non-RT tasks can only run when there are no runnable RT tasks. It selects the worker with the least total GPU load, calculated as $\sum_{q \in Q} D_q$, where D_q is the delay of query q , Q is the set of RT and non-RT queries allocated to a worker, including future RT queries.
- **Batching Queries:** A batching query can run on several workers in parallel. The master dispatches the query to a few workers and maintains the information of processed data in the batch. Before a worker running the query, it communicates with the master to get a sub-batch of the input data. The sub-batch size is controlled by the scheduler to avoid affecting future RT queries on the worker (Section 4.4.2).

4.3.3 Predicting GPU Workloads

The scheduling algorithms are based on the predicted GPU workloads on each worker. For the worker W_k , it predicts t_k^{end} and $\{Q^{future}\}_k$. t_k^{end} is the ending time of the currently running job. When the worker starts to run a job, the prediction is made as $t_k^{end} = t^{now} + D_B^M$, where D_B^M is the delay for running the DNN model M with a batch size B . $t_k^{end} = t^{now}$ for free workers.

The query set $\{Q^{future}\}_k$ consists of future streaming RT queries allocated to W_k . It is predicted based on the starting times (t^{start}), batch sizes (B) and DNN model types (M) of future queries, provided by the FutureQueries APIs. The scheduler removes a future query Q_i^{future} from the set when it starts to run the corresponding query Q_i on W_k . Then the scheduler uses t_k^{end} to represent the end time of current GPU load on the worker in the scheduling algorithms.

Estimating DNN Delays: As discussed in Section 3.3, for DNN processing delays D_B^M , the variances are reasonably small comparing to the delays. Therefore D_B^M is regarded as a constant in the estimation. To capture the drifts of D_B^M due to system fluctuations, the scheduler measures new samples D and updates the estimate as:

$$D_B^M \leftarrow \alpha \cdot D_B^M + (1 - \alpha) \cdot D. \quad (4.1)$$

If the scheduler has no previous measurement of D_B^M , it selects two nearest batch sizes B_i and B_j , which have existing delay measurements. Linear interpolation is used to obtain an estimate of D_B^M . The estimate is replaced by the measured delay after processing the DNN. The processing delay D_B^M of each video frame is varying for some tasks. For example, for the object tracking and scene graph detection tasks, the batch size B is the number of objects in one image. Figure 4.1a shows the varying batch sizes for the two tasks, which are tracked objects (detection score over 0.5) and object proposals (detection score over 0.01). The errors in estimating DNN processing delays for tracking, detection (FRCNN Inception ResNet V2) and scene graph detection tasks are given in Figure 4.1b. The errors are on the level of several

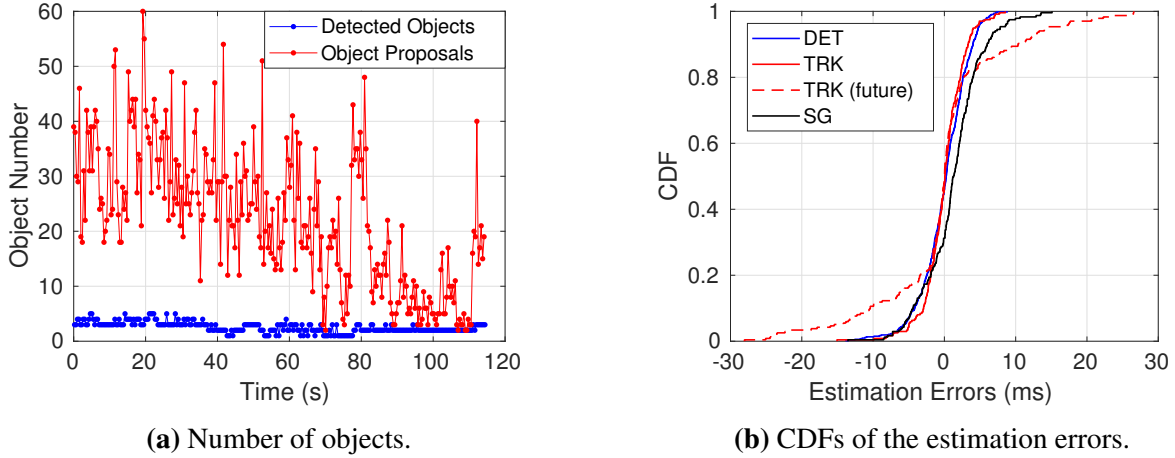


Figure 4.1. (a) Number of object boxes in a sample video stream: the numbers of objects and proposals depend on the video context, and may vary significantly. (b) DNN delay estimation performance: The varying data size causes errors in delay prediction for queries that take the data as an input batch, *e.g.*, TRK and SG queries.

milliseconds, which are reasonably low for load estimation.

To predict future load for streaming RT queries, the client needs to provide the batch sizes of future queries. For the tasks that the batch size is varying, it may introduce additional prediction errors. For example, for object tracking, we use $B_{i+1} = B_i$ by assuming that the salient objects are slowly changing between two successive frames i and $i + 1$. The number of targets may change after running a DET task. It results in errors of future query batch sizes. We give the errors of predicting D_B^M of TRK for B_{i+1} in Figure 4.1b, which are evidently larger than predicting D_B^M for B_i . These large errors only take place after DET queries running, for which the portion is P_{TRK}/P_{DET} in all TRK queries. The errors result in performance degradation in predicting GPU load and contention for the scheduler.

Network Delays and Compensations: FutureQueries APIs provide starting times t_{client}^{start} of future queries on the client side. The time when a query arrives the server t_{server}^{start} is given by $t_{server}^{start} = t_{client}^{start} + D_{net} + \delta_{clock}$, where D_{net} is the upstream network delay, δ_{clock} is the remaining clock offset between the client and the server. In this work, we estimate $D^{start} = D_{net} + \Delta_{clock}$ and obtain t_{server}^{start} accordingly for future queries. In this work we consider a cloudlet setting using

high-bandwidth Wi-Fi networks. The mobile clients are coarsely synchronized with the server using the Network Time Protocol (NTP) [106]. Using Raspberry Pi 3 boards as the mobile client ends, we measure the upstream delays offloading video frames with sizes from 21KB to 64KB. The 90 percentile is 23ms and the 99 percentile is 68ms. Consider that the network delay is relatively low, we use a lightweight method to estimate D^{start} . We measure samples D of D^{start} and use a smoothed value over the samples as the estimate $D^{start} \leftarrow \beta \cdot D^{start} + (1 - \beta) \cdot D$. We devise more accurate network delay estimation and clock synchronization methods in Chapter 5, which are used for mitigating CPU resource contention.

Dynamic Mobile Offloading: Many mobile offloading frameworks adapt offloading configurations online to tune on-board energy consumption and application performance [37, 60]. For example, a VID application may adjust the detection period or the DNN model of detection, according to battery, input video quality, and network conditions. Although DeepQuery uses the predictability of mobile workloads, it supports such online changes because it only needs prediction of the next request. When offloading configuration is changed when it submits the i th request, the changes will be applied starting from the $i+1$ th request.

4.4 Scheduling Algorithms

4.4.1 Dynamic Batching for Parallel Stages

We use a data parallelization method to process a parallel stage with a batch size B on N_{gpu} GPUs. If all GPUs are free, it is optimal to evenly distribute the workload. A GPU G_k processes a batch size $B_k = B/N_{gpu}$. However, considering the running workloads on GPUs, B_k is selected dynamically by the scheduler to minimize the processing delay. When a GPU G_k becomes free and tries to process a parallel stage, the optimal batch B_i to run on a device G_i satisfies:

$$t_i^{end} + D_{B_i} = t_j^{end} + D_{B_j} \text{ and } \sum_1^{N_{gpu}} B_i = B, \quad (4.2)$$

where D_{B_i} is the estimated delay of processing B_i , and t_i^{end} is the time when G_i will become free. Because the estimate D_{B_i} does not have a closed analytical form, to avoid expensive numerical methods, we use a linear function $D_B = k \cdot B$ to approximate it. The optimal B_i is then obtained as:

$$B_i = [(D_{B_i} + \sum_{i=1}^{N_{gpu}} t_i^{end}) / N_{gpu} - t_i^{end}] / D_{B_i} \cdot B, \quad (4.3)$$

The scheduler obtains B_k using Equation 4.3 only for the GPU G_k that is requesting a workload to run. The scheduler then updates values $t_k^{end} = t_{now} + D_{B_k}$ and $B \leftarrow B - B_k$. Equation 4.3 is re-evaluated when the next sub-batch to process is requested.

4.4.2 Dynamic Batching for Non-RT Jobs

Resource utilization can be improved by co-locating RT and non-RT jobs on shared GPUs. It requires that non-RT jobs yield GPUs when any RT jobs are ready to run. On CPUs, it is enabled by *preemptive multi-tasking*. However, without special handling in context switches enabled by driver extension, preemption incurs large overhead in both processing delay and throughput for GPU workload [142, 118]. Since such extensions are not yet generally available, this work solves the preemption problem by predicting future RT jobs and accordingly limiting the time budget to run non-RT jobs, to have them end before RT jobs starting.

When a GPU (G_k) tries to run a non-RT job with a batch size b_k , the scheduler reduces b_k if the original value may affect a future RT query. It looks up the future RT queries on each GPU G_i ($\{Q_i^{future}\}$ with a size N_{query}), and obtains a sorted list of the start times of future queries t_i^{start} , where $i \in [1, N_{query}]$. The algorithm selects the maximal b_k that does not influence any future RT query. Given that there are currently N_{nrt} running non-RT jobs, the time budget to run them are limited by the latest N_{nrt} start times (t_i^{start}). The time budget to run a new non-RT job is given as $D_{budget} = t_k^{start} - t_{now}$ if $k \leq N_{query}$, where $k = N_{gpu} - N_{nrt}$. $D_{budget} = \infty$ if $k > N_{query}$. The scheduler then reduces the batch size b_k to satisfy $D_{b_k} \leq D_{budget}$, then runs the non-RT job.

For a non-RT but non-batching job, to avoid influencing RT queries, the scheduler

estimates the time budget that a free GPU has to execute it. If the budget is not enough, instead of reducing the batch size, the scheduler keeps the job pending and the GPU idle.

4.4.3 Scaling to Multiple Workers

When there are multiple workers, the scheduler selects the best worker to dispatch each incoming query. As described in Section 5.6.3, for RT queries, when the server receives query Q_i with the information of the next query Q_{i+1} , the scheduler estimates the resource contention Φ_k after dispatching Q_{i+1} to worker W_k as follows:

$$\Phi = \int_{t_{now}}^{t_{end}} \phi(t) dt, \quad (4.4)$$

where $\phi(t)$ is a function of the number of requested GPUs $\theta(t)$ at time t , given by:

$$\phi = \begin{cases} \theta(t) - N_{gpu}, & \text{if } \theta(t) > N_{gpu} \text{ or } (\theta(t) < N_{gpu} \text{ and } \Phi > 0). \\ 0, & \text{otherwise.} \end{cases} \quad (4.5)$$

Equation 4.4 and 4.5 accumulate the contention Φ when the system is overloaded, and consume Φ when it is underloaded. The scheduler selects the worker using $k = \arg \max \Phi_k$. When there is no GPU contention on any workers, we consider two strategies: (1) selecting the worker to which the previous query of the application is allocated (contention-affinity), to avoid transferring application state data (if any) between workers; and (2) selecting the worker with the least GPU load of running and future queries (named contention-load). We also implemented a baseline method that selects the worker with minimum load, without considering contention (load). The strategies are compared in Section 4.5.3.

4.5 Evaluation

We consider the scenarios that multiple clients run applications in Figure 1.1 on local servers. We first evaluate data parallelization with dynamic batching (Section 4.5.1), and then demonstrate the effectiveness of co-locating RT and non-RT queries (Section 4.5.2), as well as the algorithms that dispatch queries to workers (Section 4.5.3).

4.5.1 Dynamic Batching for Parallel Stages

To evaluate data parallelization and dynamic batching, we set 4 clients running the VID application on 4 GPUs, with two sets of configurations: $P_{TRK} = 160\text{ms}$, $P_{DET} = 1.6\text{s}$, and $P_{TRK} = 200\text{ms}$ and $P_{DET} = 2\text{s}$. We test two FRCNN models, Inception ResNet V2 and NasNet, using 300 proposals, *i.e.*, $b = 300$. As described in Section 3.3.5, a DET task comprises of tracking, detection, and result fusion. We use the end times of result fusion to obtain DET delay. We use the query processing delay on servers as the performance metric. It includes server queuing and processing delays, but excludes network delays between clients and servers.

The comparisons include three baselines: (1) no parallelization; (2) a fixed batch size $b_i = 30$ and (3) $b_i = 75$. As shown in Figure 4.2a. We observe that the parallelization effectively improves delays of both TRK and DET queries. For both models, although a smaller batch size ($b_i = 30$) means finer grains of workload balance on multiple GPUs, it decreases the processing throughput. As a result, it has longer TRK and DET delays compared to $b_i = 75$. Dynamic batching control evidently improves the DET delay for both models. For NasNet, we observe that the TRK delay becomes larger than $b_i = 75$ when dynamic batching is used. It is because dynamic batching may allocate larger batches (> 75) of the CLS stage of FRCNN that compete GPU resource with TRK jobs. The result demonstrates that dynamic batching is effective in reducing delays for jobs with parallel stages.

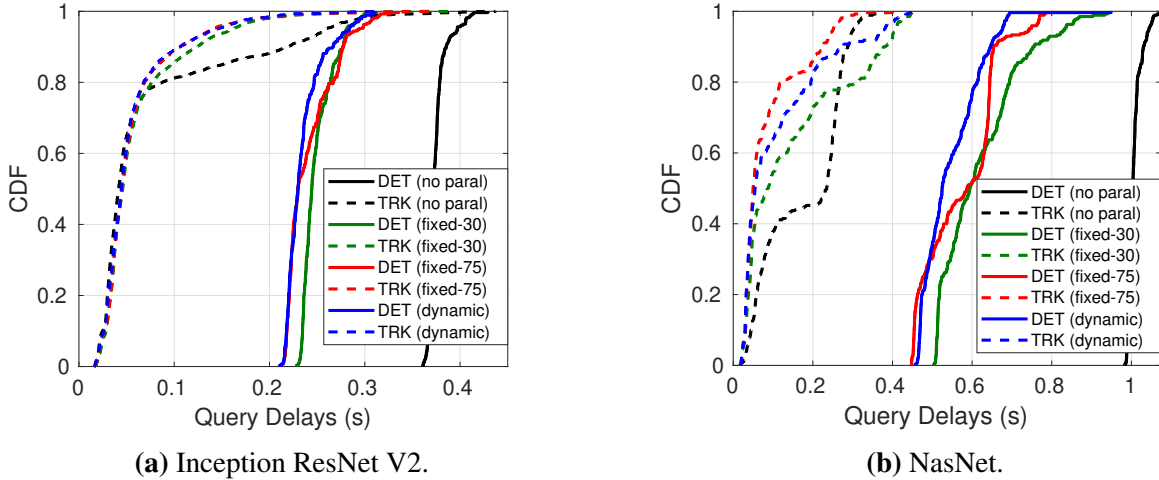


Figure 4.2. CDFs of DET and TRK query delays, using different batching strategies for the parallel CLS stage of FRCNN. It shows that using a fixed small batch reduces throughput, however, a fixed large batch can not be fully parallelized. The dynamic batching method selects batch size according to running jobs on all GPUs to improve parallelism and reduce delay.

4.5.2 Co-locating RT and Non-RT Queries

To evaluate the method for co-locating RT and non-RT queries, we process both types of queries on shared GPUs, and measure the influence on the delays of RT queries.

Batching Queries: We evaluate the dynamic batching algorithm for non-RT jobs using a ResNet-152 feature extractor as an example. The algorithm should use batch sizes as large as possible to increase the throughput of ResNet-152, while avoiding affecting RT queries. A client runs a VID application (RT) on a GPU, with a low overhead configuration $P_{TRK} = 400\text{ms}$, $P_{DET} = 4\text{s}$. Another client submits a non-RT query extracting ResNet-152 features from a batch of 1024 images. The batching query may result in larger delays of VID queries. As the baselines, we use a fixed batch size 1, 4, 6, 8 and 16 to process the ResNet-152 query. As shown in Figure 4.3, a small batch size $b = 1$ results in a large processing delay of CNN, because of a low throughput. However, a large batch size $b = 16$ results in larger delays of TRK jobs, because it may occupy the GPU when a TRK is ready to run. The result shows that the dynamic batching methods can improve the throughput of non-RT jobs while maintaining low TRK delays. Although selecting an appropriate fixed batch size helps improve the performance as well, the value needs to be

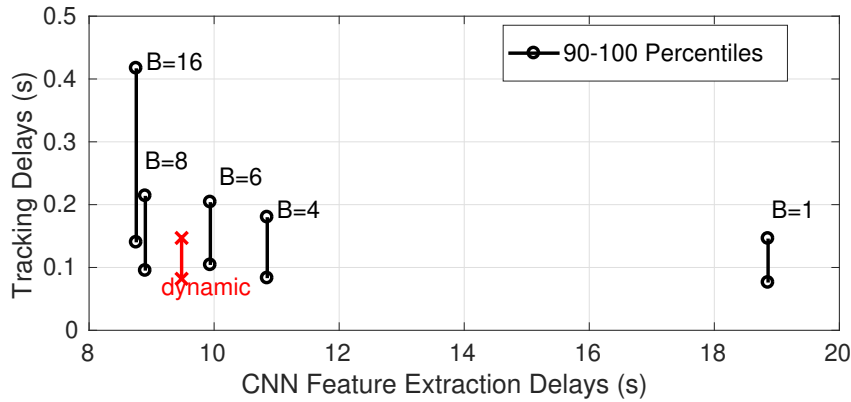


Figure 4.3. Delays of CNN (non-RT) and delay tail percentiles of TRK (RT) queries, comparing different batching strategies. The dynamic batching method decides the maximal batch size that does not affect the RT jobs, which improves the delays of both RT and non-RT jobs.

selected manually which is undesired.

The time slots of jobs on GPU is shown in Figure 4.4. The RT queries (DET, TRK) are plotted in red, and ResNet-152 is in yellow. It illustrates how a large batch size influences TRK jobs. Our algorithm selects the batch sizes dynamically to have non-RT jobs end before future RT jobs.

Non-Batching Queries: For a non-RT query with a non-batching input, the scheduler runs it only when it does not affect future RT queries. We use scene graph queries processing 1 image as the example. The interesting part is that the delay of SG depends on the number of object proposals. It is varying frame to frame in a video. The scheduler estimates the delay of a SG task to decide whether it can run.

In the experiments, a client running VID submits SG queries together with DET queries. SG takes the object proposals obtained from DET as its input. The delays of SG queries are from server receiving the image to obtaining scene graphs, including the delays of DET. The experiments have 2 clients and one of them submits SG queries. As shown in Figure 4.5, SG queries introduce additional delays to TRK and DET queries, which are originally 370ms (TRK) and 429ms (DET). When one client submits a SG query with each DET query, without scheduling SG as non-RT queries, TRK and DET delays increase to 471ms and 489ms. The scheduling

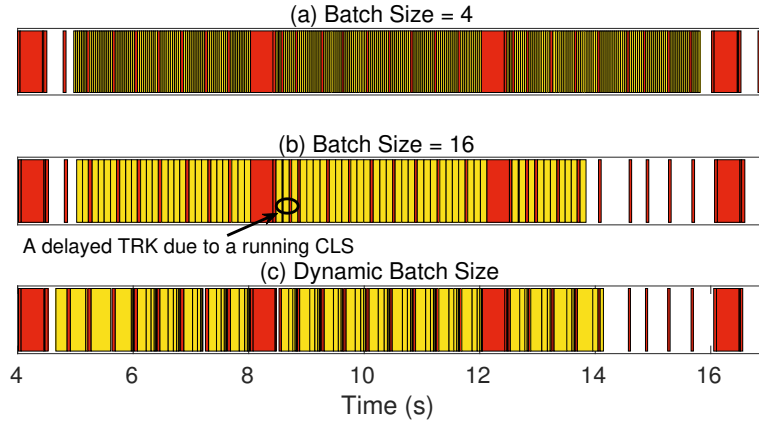


Figure 4.4. Time slots of DET, TRK (in yellow) and CNN jobs (in red) on one GPU. Using a small batch size for CNN (4) leads to a low processing throughput and a long delay, as shown by the total makespan of the yellow slots. A large batch size (16) results in significant contention with RT jobs: the long duration of the yellow slots affects the start times of the red slots. It illustrates how the dynamic batching method selects the batch size for non-RT jobs according to the time budget to complete before RT jobs.

method reduces the delays to 374ms (TRK) and 452ms (DET). In this case, because SG tasks can only run when there is enough time budget, their processing delays increase dramatically.

4.5.3 Worker Selection Strategies

To evaluate the algorithms to dispatch incoming queries to workers, Figure 4.6 compares the delays of DET and TRK queries when 4 clients run on a worker with 4 GPUs, and another setting with two workers, each with 2 GPUs. Object detection uses the Inception ResNet V2 FR-CNN model. Because data parallelization causes difference in delays for workers with different number of GPUs, it is not used to do a fair comparison. The results show that the one worker setup has better performance, the tail delays of DET and TRK are 429ms and 437ms respectively. It is because there is no workload balance issues in this case. However, when distributing workloads on multiple workers, the imbalance of workloads may reduce the utilization of GPUs. Although the scheduler aims to balance GPU loads by minimizing contention, the prediction errors of network delays and DNN processing delays result in remaining contention. The result shows that the contention-load approach has the minimal tail DET and TRK delays, which are

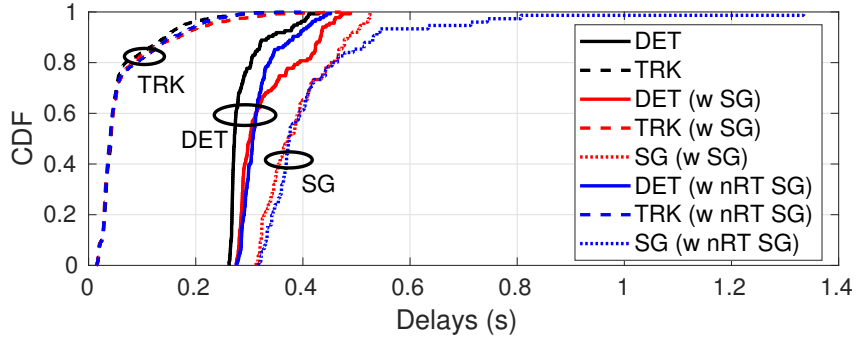


Figure 4.5. CDFs of DET, TRK and SG serving 2 clients, including cases of no SG queries, and one client submitting SG queries (with and without scheduling SG as non-RT jobs). It illustrates that when SG queries are scheduled as non-RT, they yield resource to RT queries and become slower to complete. As a result, RT queries become faster as observed from the delay CDFs.

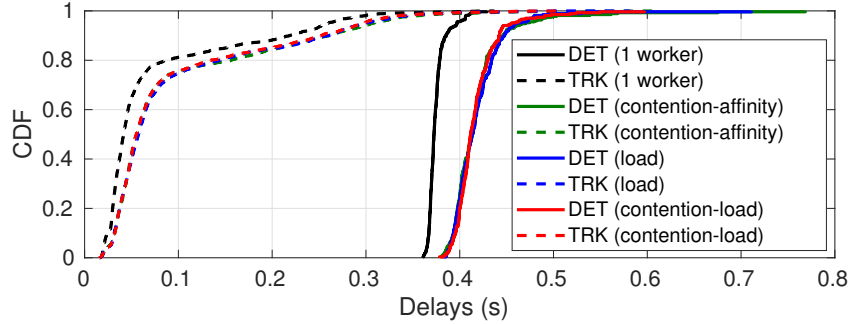


Figure 4.6. CDFs of DET and TRK query delays serving 4 clients, including having 1 worker with 4 GPUs, and 2 workers with 2 GPUs using different scheduling strategies: contention-affinity, load, contention-load. It shows that the delays are larger when the computing resources are distributed onto two machines, due to possible load imbalance. Contention-load performs the best in selecting worker servers for queries, which achieves the smallest tail delays.

597ms and 512ms. The CDFs of delays are similar for contention-affinity and load, but they have much larger tail query delays.

4.6 Summary

In this chapter, we present the methods that increase GPU resource utilization by co-locating RT tasks and non-RT tasks on shared GPUs, on local servers in a cloudlet. The resource contention on GPUs can be alleviated by controlling the batch sizes of non-RT tasks dynamically, based on predictions of GPU resources requested by RT tasks. We demonstrate the usage and

effectiveness of the schedulers through studying real world applications and a variety of DNN models.

Chapter 5

Mitigating Resource Contention on CPUs

In this chapter, we study how to mitigate resource contention for tasks running on CPUs to provide users a satisfactory experience with low delays. We present a new Plan-Schedule method that adjusts task start time ahead of execution by predicting future contentions. The prediction capability is enabled by task execution time prediction, network delay estimation, and clock synchronization between the client and the server.

5.1 Introduction

Workload scheduling is essential for RT vision tasks running on CPUs to attain low delays. On multi-tenant computing servers without dedicated scheduler algorithms, scheduling is usually either delegated to the server OS or done using the First-Come-First-Serve (FCFS) strategy. The processor resource contention increases workload processing delays in the former approach, while FCFS causes additional queueing delays. To improve E2E delays, methods for scheduling workloads with timing requirements have been extensively studied, *e.g.*, RT tasks with deadlines [94] and big analytics tasks [146, 127]. Task reordering is adopted in these schedulers: When resource contention takes place, the reordering method prioritizes the most urgent tasks and postpones the others. We observe that reordering does not mitigate resource contentions. It is a workaround to meet task timing needs by using time slack of other tasks.

In this chapter, we present methods to attain low delays in a cloudlet for workloads

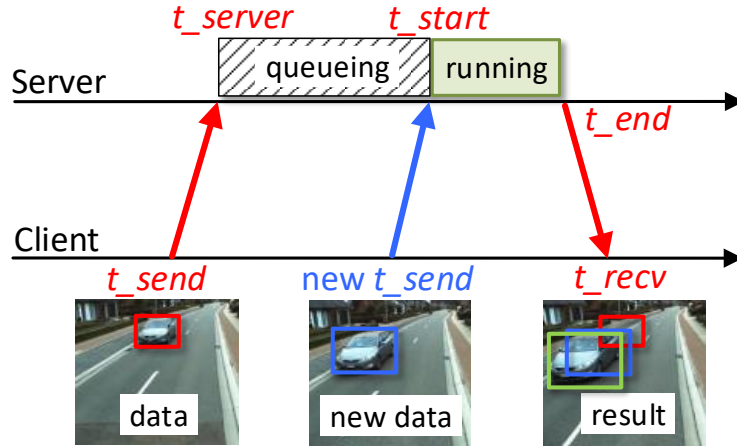


Figure 5.1. ATOMS predicts processor contention on multi-tenant servers and postpones the start time (t_{send}) of an offloaded task to avoid its queueing delay.

running on CPUs. The servers may be heavily loaded due to lack of dynamically provisioned resources. Our methods mitigating CPU contention instead of reordering tasks are implemented into a scheduler called *ATOMS* (Accurate Timing prediction and Offloading for Mobile Systems).

The basic idea of the scheduling in *ATOMS* is shown in Figure 5.1 through an example of offloading a detection task to a server which adopts the FCFS task scheduling. If the queueing caused by multi-tenancy can be predicted, the mobile client can postpone offloading and get a more recent result (blue box) that better localizes the moving car (green box). We develop a two-phase *Plan-Schedule* strategy to realize the idea. In the planning phase, *ATOMS* predicts time slots of future tasks from all clients, detects contention, coordinates tasks, and informs clients about the new offloading times. In the scheduling phase, for each arriving task, *ATOMS* selects a machine to execute it which has the minimal estimated contention. We demonstrate the effectiveness of *ATOMS* in Figure 5.2. 4 clients offload neural network object detection tasks to a 8-core server. Figure 5.2a shows the time slots of offloaded tasks using FCFS scheduling. The load line (red) gives the total number of concurrent tasks, and contention (queueing) takes place when the load exceeds 1. Figure 5.2b shows that contention is almost eliminated because of the dynamic load prediction and management by *ATOMS*.

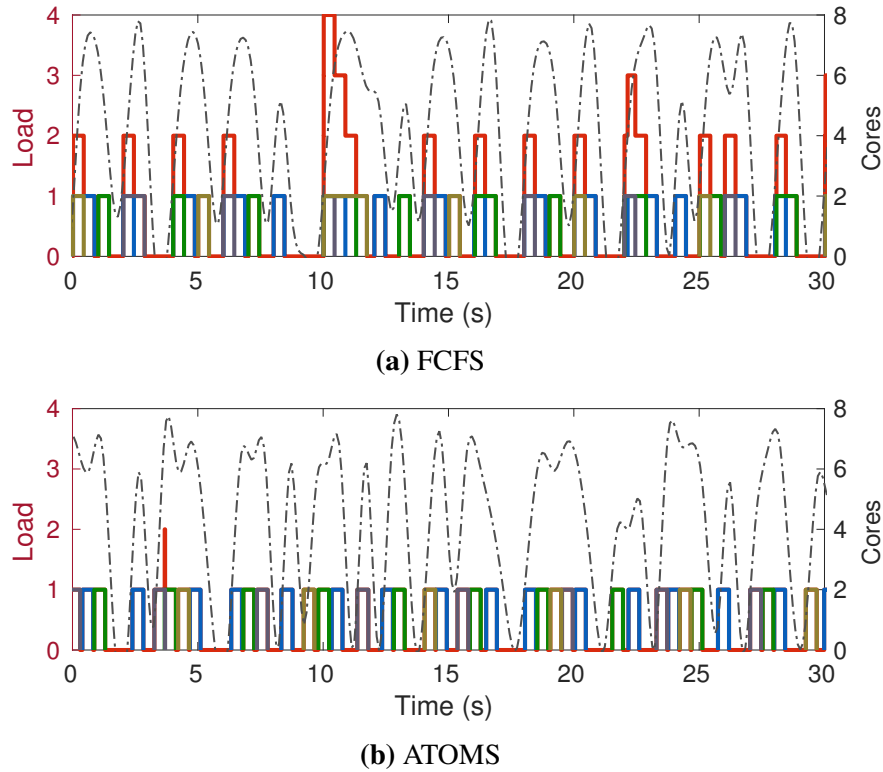


Figure 5.2. 4 clients offload neural network object detection to a 8-core server with periods 2s to 5s. The time slots of offloaded tasks are plotted and the load line (red) gives the total number of concurrent tasks. The dashed curve gives the CPU usage (cores) of the server.

In implementing ATOMS, we leverage the correlation between frames in continuous video data from a camera to estimate the computing time which may vary significantly, through time series prediction. The contention detection and task coordination across all mobile clients are enabled by the estimation of wireless network latency as well as its uncertainty, and client-server clock synchronization. We support Service Level Agreements (SLAs) specified by clients to control the deviation of offloading interval from the desired period, which is caused by dynamic task coordination. ATOMS deploys applications in containers, which are more lightweight and flexible than VMs, to hide the complexities of diverse programming languages, runtime systems, and dependencies. The framework can be deployed in both conventional cloud computing environments as well as cloudlets. We demonstrate the performance improvement of offloading systems with ATOMS under public campus WiFi and LTE networks in the evaluations.

5.2 Background

Real-Time Schedulers: Deadline based schedulers, *e.g.* Earliest Deadline First (EDF) scheduler [94], minimize deadline miss rates rather than E2E delays, so they are not ideal to solve our problem. A simple FCFS scheduler is closer to our need: it prioritizes the task with the longest waiting time and thus improves tail E2E delays. Both EDF and FCFS schedulers are based on reordering or preemption that leave other tasks waiting on the contended resources. Compared with them, ATOMS achieves lower E2E delays by avoiding queueing to further reduce E2E delay.

Resource reservation is another effective way to schedule recurring RT workloads [124]. Previous approaches that make periodical resource reservations statically are limited for a general mobile offloading scenario: clients can adjust the offloading rate to tune the performance and power consumption, workload processing times may vary significantly over short time scales, and network latency spikes may invalidate such reservations. ATOMS leverages estimated task computing times and network delays to make dynamic reservations that precisely fit incoming workloads.

Cloud Schedulers: Cloud schedulers have been well engineered to handle a wide range of data processing jobs with time-based SLAs. These systems leverage rich information, for example, estimates and measurements on resource demand and running time, to reserve and allocate resources, and to reorder tasks in queue [146, 127]. Because data processing tasks have makespans and deadlines in much larger time scale, usually ranging from seconds to hours, these methods are inadequate for handling real-time mobile applications that require sub-second delays.

5.3 System Model and Performance Metrics

We denote a mobile client as C_i with an ID i . The offloading server is a distributed system composed of resource-rich machines. An offloading request sent by C_i to the server is

denoted as task T_j^i , where the task index j is monotonically increasing. We omit the superscript for simplicity when discussing only one client.

Figure 5.1 shows the life cycle of an offloaded task. T_j is sent by a client at t_send_j . It arrives at a server at t_server_j . After queueing, the server starts to process it at t_start_j and finishes at $t_end_j = t_start_j + d_compute_j$, where $d_compute_j$ is the computing time.¹ The client receives the result back at t_recv_j . T_j uses T_paral_j cores in parallel. Because a parallel program may have sequential sections that can not fully utilize all cores, the parallelism T_paral_j can be a non-integer.

We evaluate a task using two primary performance metrics of continuous mobile sensing applications [79]. **E2E delay** is calculated as:

$$d_delay_j = t_recv_j - t_send_j, \quad (5.1)$$

which comprises the upstream network delay d_up_j , queueing delay d_queue_j , computing time $d_compute_j$ and downstream delay d_down_j . ATOMS minimizes d_queue_j to reduce d_delay_j .

Offloading interval represents the time span between successive offloaded tasks of a client, calculated as:

$$d_interval_j = t_send_j - t_send_{j-1}. \quad (5.2)$$

Smaller interval can help improve the application performance, but would exert higher power consumption on the client and more workloads on the server. In ATOMS, C_i sends offloaded tasks periodically, ideally any interval is equal to d_period_j . However, the interval is not constant due to task coordination. We desire a smaller interval jitter that provides stable sensing activities, which is given by:

$$d_jitter_j^i = d_interval_j^i - d_period_j. \quad (5.3)$$

¹A symbol starting with “ $t_$ ” is a timestamp and “ $d_$ ” is a time duration.

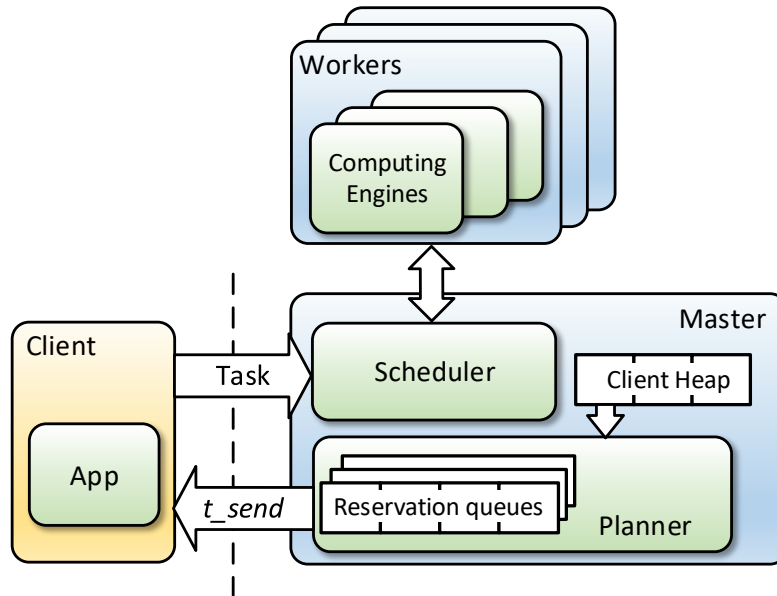


Figure 5.3. The architecture of ATOMS framework. The planner predicts the time slots of future offloaded tasks from all clients. It predicts future resource contention and adjusts the start times of future tasks to mitigate contention. After adjusting the start time of a task, ATOMS uses the adjusted time (t_{send}) to send a task to the client. The task is sent at t_{send} and is then scheduled to run on one worker server.

Clock synchronization (see Chapter 5.5.4) is employed to validate the calculation across client-side and server-side timestamps.

5.4 ATOMS System Design

In this section we present the framework design and detail the plan-schedule algorithms. The architecture of ATOMS is shown in Figure 5.3. It is composed of one *master* server and multiple *worker* servers. The master communicates with the clients and dispatches tasks to workers for execution. It is responsible for planning and scheduling tasks. Deploying and running applications on workers have been discussed in Section 3.6.2.

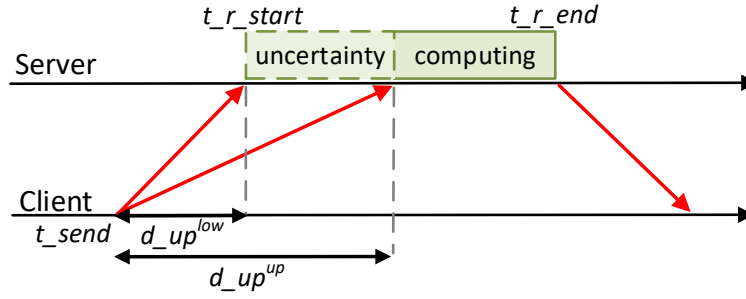


Figure 5.4. Processor reservation for future offloaded task considering the uncertainty of arriving time at server and its computing time. The uncertainty of arriving time includes the variation of network upstream delay and the clock offset between the client and the server.

5.4.1 Master and Offloading Workflow

In addition to the basic send-compute-receive offloading workflow, ATOMS has three more steps: *creating reservation*, *planning*, and *scheduling*.

Reservation: We first introduce *reservation*, the processor resource that a future offloaded task requires. When the master starts to plan task T_j , it creates a new reservation $R_j = (t_{r_start_j}, t_{r_end_j}, T_{paral_j})$, where $t_{r_start_j}$ and $t_{r_end_j}$ are the starting and ending time, respectively, and T_{paral_j} is the number of cores requested. As shown in Figure 5.4, with the lower and upper bounds of upstream network latency ($d_{up_j}^{low}$, $d_{up_j}^{up}$) estimated by the master (Section 5.5.2), as well as the predicted computing time ($d_{compute'_j}$), the span of reservation is calculated as:

$$t_{r_start_j} = t_{send_j} + d_{up_j}^{low}, \quad (5.4)$$

$$t_{r_end_j} = t_{send_j} + d_{up_j}^{up} + d_{compute'_j}. \quad (5.5)$$

The time slot of R_j contains the uncertainty of the time when T_j arrives at server (t_{server_j}), and the computation time consumed. Given that the predictions of network bounds and computing time are correct, each task falls into the reserved time slot.

Planning: The planning phase takes place before the real offloading happens. It coordinates

future tasks of all clients to ensure that the total amount of all reservations never exceeds the limit of total processor resources on all workers. We call the component that runs the planning algorithm as *Planner*.

Client C_i registers at the ATOMS master to initialize the offloading process. Master assigns it a future timestamp t_send_0 indicating when to send the first task. Master creates a reservation R_j for task T_j and start to plan it as soon as:

$$t_{now} + d_future > t_r_start_j, \quad (5.6)$$

t_{now} is the current clock time of the master. d_future is a parameter on how far after t_{now} that the planning covers. The planner predicts and coordinates future tasks in the time range from t_{now} to $t_{now} + d_future$.

We denote the next task of client C_i to plan as T_{next}^i . The master plans future tasks chronologically by their starting times $t_start_{next}^i$. It keeps all clients in a min heap with $t_start_{next}^i$ as the keys and checks whether a new task should be planned using Equation 5.6. If so, provided that the top of the heap is client C_i , a new reservation R_{next}^i is created. The next task to plan of C_i is updated to T_{next+1}^i with the sending time:

$$t_send_{next+1}^i = t_send_{next}^i + d_period_i. \quad (5.7)$$

Planner takes the new reservation R_{next} as input. It detects resource contention and then resolves it by adjusting the sending time of both the new task being planned and a few planned tasks. We give the details of the planning algorithm in Section 5.4.2. If the sending time of the last planned task T_{next-1} of any client has been adjusted for Δ_{next-1} , in order to maintain the period $t_send_{next} = t_send_{next-1} + d_period$, it adjusts the sending time of T_{next} accordingly:

$$t_send_{next} \leftarrow t_send_{next} + \Delta_{next-1}. \quad (5.8)$$

And a reservation R_j^i remains adjustable until:

$$t_{now} + d_inform_i > t_send_j^i, \quad (5.9)$$

d_inform_i is a parameter of C_i on how early the master should inform it about the adjusted task sending time. The planner then removes R_j and notifies the client. Upon receiving t_send_j , the client sets a timer to offload T_j . d_inform should be larger than the downstream latency to deliver it on time. On the other hand, a large d_inform causes reservations to be removed and become unadjustable too early, which limits the capability of planning.

Scheduling: The client offloads T_j to the master when the timer at t_send_j timeouts. After receiving the task, using the information of tasks currently running on each machine, the scheduler selects the machine which induces the least processor contention. The master dispatches it to the worker machine and returns the result. We give its details in Section 5.4.3.

5.4.2 Planning Algorithms

The planner buffers reservations in *reservation queues*. A reservation queue stands for a portion of processor resource in the cluster. A queue Q_k has a resource limit of Q_cpu_k cores for contention detection. The sum of Q_cpu of all queues is equal to the total number of cores in the cluster: $\Sigma Q_cpu = \Sigma W_cpu$. Each computing engine is associated with a reservation queue. A reservation can only be inserted to the queues that support it, which have engines for that application. In the planning process, the parallelism of a reservation $T_parallel$ is determined by the processor limit of computing engines. For example, $T_parallel$ of a fine-parallelized task is different for an engine on a dual-core worker ($T_parallel = 2$) and one on a quad-core worker ($T_parallel = 4$).

Contention Detection: When the planner receives a new reservation R_{new} , it first decides which queue to place it in. It iterates over all the queues, for Q_k , and calculates the needed amount of time (Δ_k) to adjust R_{new} and the total processor usage (Θ_k) of Q_k during the time slot

Algorithm 1: Calculating Reservation Adjustment

Input: Σ_R : all reservations in the queue Q ;
 R_{new} : the new reservation
Output: Δ : the adjustment of R_{new} to remove contention

- 1 Initialize the empty list δ_list , $\Delta = 0$
- 2 $\delta_{new}^{start} = new_delta(t = t_r_start_{new}, v = T_paral_{new})$
- 3 $\delta_list.add_elements(\delta_{new}^{start})$
- 4 **for** $R_j \in \Sigma_R$ **do**
- 5 $\delta_j^{start} = new_delta(t = t_r_start_j, v = T_paral_j)$
- 6 $\delta_j^{end} = new_delta(t = t_r_end_j, v = -T_paral_j)$
- 7 $\delta_list.add_elements(\delta_j^{start}, \delta_j^{end})$
- 8 Sort δ_list by $\delta.t$ in ascending order
- 9 **for** δ_j that $\delta_j.t > \delta_{new}^{start}.t$ and $\delta_j.v < 0$ **do**
- 10 $load_{max} = \sum_{i=0}^m \delta_i.v$, $m = \arg \max_i (\delta_i.t \leq \delta_{new}^{start}.t)$
- 11 **if** $load_{max} > Q_cpu$ **then**
- 12 $\Delta = \delta_j.t - t_r_start_{new}$
- 13 $\delta_{new}^{start}.t = \delta_j.t$
- 14 **else**
- 15 **break**
- 16 **return** Δ

of R_{new} . The planner selects the queue with the minimal Δ .

Algorithm 1 describes how to calculate Δ , where δ stands for a change of processor load, with t being the time and v being the amount. It checks whether the total load on Q_k after adding R_{new} exceeds the limit Q_cpu . If so, the algorithm checks the end times of reservations in queue to calculates Δ_k that the contention can be eliminated after postponing R_{new} by Δ_k . Otherwise it returns $\Delta_k = 0$. We give an example in Figure 5.5 where a new reservation R_0^2 is being inserted into a queue. The solid black line in the lower plot is the total load $load(t)$ and all δ s are plotted as arrows. Contention arises after adding R_0^2 . It can be removed by postponing R_0^2 to the end time of R_1^1 . Δ is thus obtained.

If two or more planning queues have the same Δ , for example, when multiple queues are contention-free ($\Delta = 0$), the planner calculates the processor usage Θ during the time slot of R_{new} :

$$\Theta = \int_{t_r_start_{new}}^{t_r_end_{new}} load(t) dt. \quad (5.10)$$

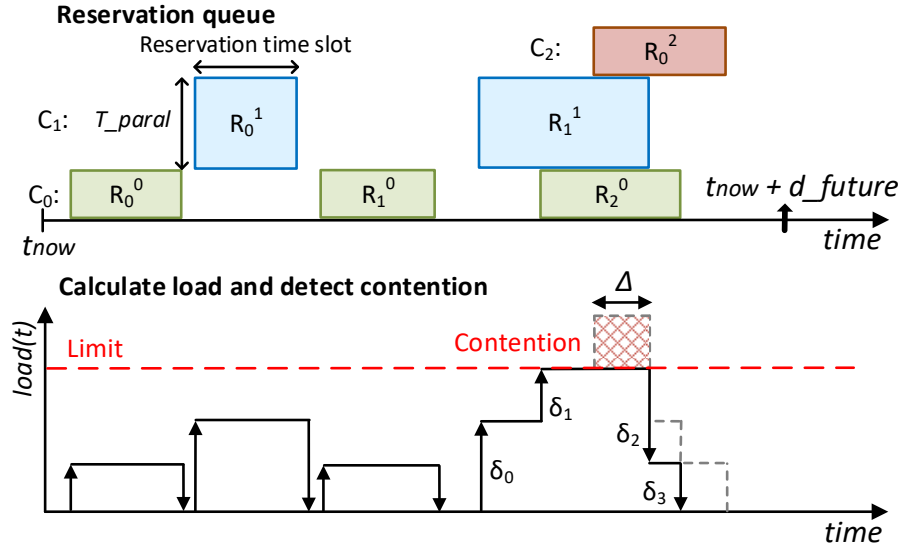


Figure 5.5. An example of detecting processor contention and calculating required reservation adjustment. The top plot shows a reservation queue containing reservations for different clients. The bottom plot shows the calculated total load $load(t)$. Contention is detected when $load(t)$ exceeds the number of CPU cores.

We consider two strategies. The *Best-Fit* strategy selects Q that has the highest Θ , which leaves the least margin of processor resources on the queue and packs reservations as tightly as possible. The other strategy is *Worst-Fit* that, in contrast, selects the queue with the lowest Θ . We study their difference through evaluations in Section 5.6.4.

SLAs: In the next *coordinating* step, instead of simply postponing R_{new} by Δ , the planner moves ahead a few planned reservations as well, to reduce the amount of time that R_{new} has to be postponed. The coordinating process takes Δ as input and adjusts reservations according to *cost* (R_{cost}), a metric on how much the measured offloading interval $d_{interval}$ deviates from the client's SLAs (a list of desired percentiles of $d_{interval}$). For example, a client with period 1s may require a lower bound $d_{slo}^{10\%} > 0.9s$ and an upper bound $d_{slo}^{90\%} < 1.1s$.

The master calculates R_{cost} when it plans a new task, using the measured percentiles of interval ($d_{interval}^P$). For the new reservation (R_{new}) to be postponed, the cost R_{cost}^+ is

obtained from the upper bounds:

$$R_cost^+ = \max_{p \in \cup^+} (\max(d_interval^p - d_slo^p, 0)), \quad (5.11)$$

where p is a percentile and \cup^+ is the set of percentiles that have upper bounds in the SLAs. R_cost^+ is the maximal interval deviation from SLAs. For tasks to be moved ahead, deviation from lower bounds (\cup^-) are used instead to obtain the cost R_cost^- . The cost is a relative weight among all the clients. SLAs that contain tight bounds on $d_interval$ make tasks of the client less affected during the coordinating process.

Reservation Coordination: We adopt *reservation block* (B) as the basic unit of coordination to simplify the process. The slack between two reservations R_i, R_j where $i < j$ is defined as $slack_{i,j} = t_r_start_j - t_r_end_i$. Any two reservations with no slack in between are in the same block. A block's start time t_b_start is when the first reservation starts and its end time t_b_end is when the last reservation ends. Each reservation queue keeps an associated queue of blocks, sorted by t_b_start . The cost of a block B_cost is the maximal cost of all its contained reservations. All reservations in a block B_i are adjusted by the same amount of time during coordination.

Algorithm 2 describes the coordinating process. It starts with the last two blocks, and iteratively adjusts two adjacent blocks in reverse order. The amount of time to move away two blocks, initialized to Δ (given by Algorithm 1), is divided proportionally according to B_cost . It is equally divided if both costs are zero. Moving a block ahead reduces the slack between itself and the previous block. Two blocks are merged when the slack becomes zero. The process ends if B_i has enough slack to move ahead or it is already the first block in queue. After obtaining the adjustment of reservation R_j , the change is applied to the sending time (t_send_j) of the associate task that will be sent to the client. Figure 5.6 gives an example: R_1^1 is the new reservation that forms a new block B_3 . We first move B_2 and B_3 apart and merge them. Then B_1 is moved forward and merged with R_1^1 and R_2^0 . The process ends because there is enough slack between B_0 and B_1 .

Algorithm 2: Coordinating Reservations

Input: B_list : a sorted list of reservation blocks;
 R_{new} : the new reservation;
 Δ : the initial adjustment given by Algorithm 1

- 1 Create a new block B_{new} containing R_{new}
- 2 **for** $B_i \in B_list \cup \{B_{new}\}$ **in reverse order do**
- 3 **if** $B_i \neq B_{new}$ **then**
- 4 $\Delta = -(t_b_start_i - t_b_end_{i-1})$
- 5 **if** $\Delta \leq 0$ **then**
- 6 **break**
- 7 **else**
- 8 $\Delta_{i-1} = -B_cost_i^+ / (B_cost_{i-1}^- + B_cost_i^+) \cdot \Delta$
- 9 $\Delta_i = B_cost_{i-1}^- / (B_cost_{i-1}^- + B_cost_i^+) \cdot \Delta$
- 10 Adjust B_{i-1} by Δ_{i-1} and B_i by Δ_i
- 11 Merge blocks B_{i-1}, B_i

Global/Partitioned Planning: Reservation block simplifies coordinating, however, the approach is pessimistic that it stops moving two reservations closer once the slack becomes zero. If the processor limit Q_cpu is large enough to accommodate two or more reservations in parallel, the approach results in unnecessary adjustment and larger interval jitter. For the purpose of reducing jitter, we use partitioned planning for reserved workers, which has a reservation queue for each computing engine. Multiple queues are thus associated with the worker, in order to reduce Q_cpu of each queue. For shared workers, because resource contention may happen between engines, we have to use a single reservation queue for each worker ($Q_cpu = W_cpu$). For example in Figure 3.5, we must have a reservation queue with $Q_cpu = 2$ to accommodate the FeatureMatch tasks. The three engines are all associated with that queue. We discuss assigning computing engines to reservation queues with more examples in Section 5.6.

5.4.3 Scheduling

The scheduler dispatches offloaded tasks that arrive at the master to the most suitable worker machine to minimize processor contention. The diagram of the scheduler is given in Figure 5.7. It keeps a global FIFO task queue for each application to buffer tasks when all

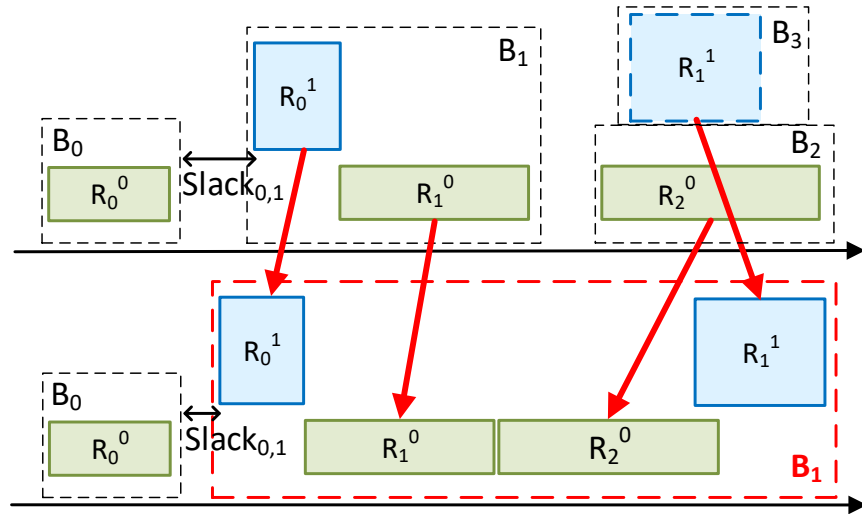


Figure 5.6. An example of coordinating reservations. A new reservation R_1^1 is to be placed in a reservation queue with three blocks. Reservations are adjusted using Algorithm 2 to remove resource contention. As result, reservation R_1^1 is postponed and R_0^1, R_1^0, R_2^0 are moved ahead.

computing engines are busy. For each shared worker, there is a local FIFO queue for each application that it serves.

Scheduling workflow: When a task arrives, the scheduler first checks available computing engine on any reserved workers. It dispatches the task if one is found and the scheduling process ends immediately. If there is no reserved worker, or no computing engine is available, the scheduler turns to check shared workers which serve the application. It selects the best worker to dispatch based on processor contention Φ and usage Θ . The task is then dispatched to a free engine on the selected shared worker. If no engine is free on the worker, the task is appended at the worker's local task queue. The task is put into the global queue if there is no shared worker for the application.

After completing the previous task, engines on shared workers get a task to process from the local task queue. For engines on a reserved worker which has no local queue, it first tries to steal a task from any shared worker's local queue. If still no task is found, it then fetches one task from the global task queue. An engine that fails to run a task becomes free and waits for

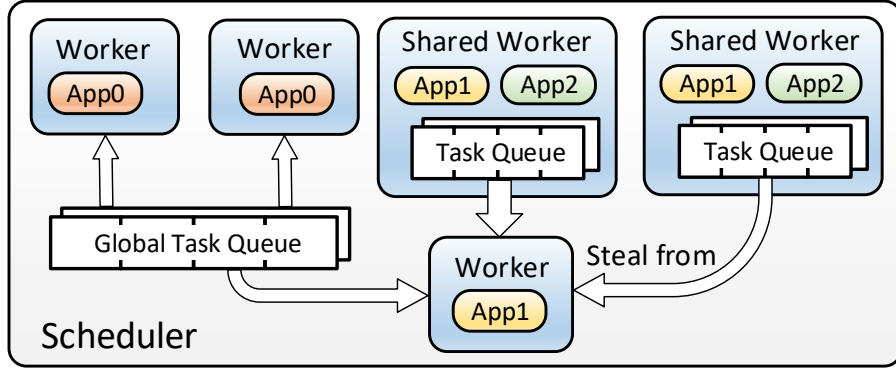


Figure 5.7. Diagram of the ATOMS scheduler. Reserved workers can steal tasks from shared workers’ local task queues and the global queues. A shared worker only gets tasks to process from its local task queue.

incoming tasks.

Worker selection: Here we detail the policy to select shared worker. T_{new} is the task to schedule. Scheduler records the estimated ending time t_{end}' of all the running tasks on each worker, using the real task starting time and the predicted computing time $d_{compute}'$. It calculates the load on each worker including the task to be scheduled using Equation 4.5. The calculation is similar to planning, the difference is that we use time slots of running tasks here instead of reservations. The resource contention Φ_k of worker W_k is calculated by:

$$\Phi_k = \int_{t_{now}}^{t_{end}'_{new}} \max(load(t) - W_{cpu_k}, 0) \cdot dt. \quad (5.12)$$

Use Figure 3.5 as an example: the red shadow region standards for the resource contention after dispatching the incoming FeatureMatch task to the worker. Its area is equal to Φ . The worker with the smallest Φ is selected to run T_{new} . For workers with identical Φ , similar to the planning algorithm (as Equation 5.10), we use processor usage Θ to as the selection metric. We compare the two selecting methods, Best-Fit and Worst-Fit, through evaluations in Section 5.6.

5.5 Implementation

We have implemented the ATOMS master in Go with 4K lines of code. The ATOMS client is implemented in C++ on Linux and in Java on Android devices. All distributed components (client, master, worker) communicate via *gRPC* [8]. In this section we present the implementation of computing time measurement and prediction, upstream network latency estimation, and clock synchronization.

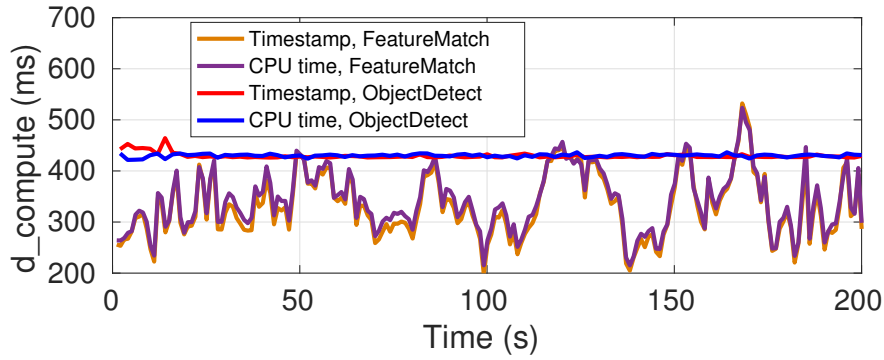
5.5.1 Measuring Computing Time

Measure computing time: The workers measure the computing time of each task, denoted as $d_compute$, and return them to the master along with the computation result. The measurements are used to predict $d_compute$ for future tasks (Section 5.5.2).

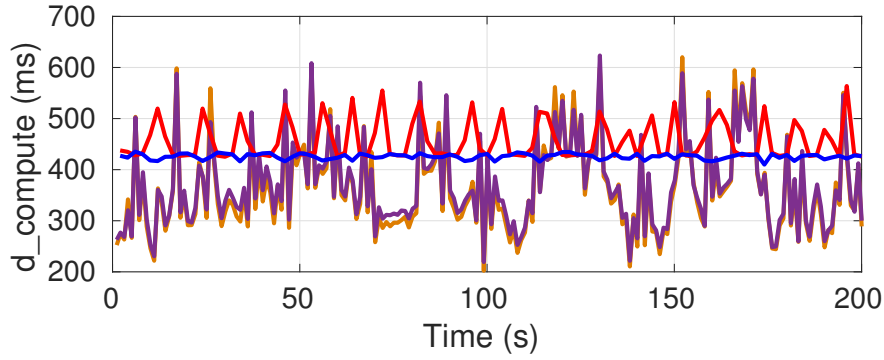
A straightforward method is measuring the starting and ending timestamps of a task, and calculating the difference ($d_compute_{ts}$). However, this method is vulnerable to processor sharing, which may happen on shared workers. $d_compute_{ts}$ increases when a task yields processor to others, which leads to inaccurate measurement. We instead get $d_compute$ through measuring *CPU time* consumed by the engine container during the computation, written as $d_cputime$. The computing time is obtained by:

$$d_compute_{cpu} = d_cputime / T_paral. \quad (5.13)$$

The OS stops counting CPU time when a process turns to sleep, so only the running time of computing engine is measured. We demonstrate the robustness of $d_compute_{cpu}$ against processor sharing through an example. Figure 5.8a gives computing time of FeatureMatch and ObjectDetect that run separately, measured by timestamps and CPU time, respectively, on a 8-core machine (Intel i7-4790, 3.6GHz). We use $T_paral = 1.7$ for FeatureMatch and $T_paral = 7.3$ for ObjectDetect to fit Equation 5.13 to make $d_compute_{cpu}$ accord with $d_compute_{ts}$. In Figure 5.8b



(a) Running two applications separately.



(b) Running two applications concurrently.

Figure 5.8. Measure computing time using timestamps ($d_compute_{ts}$) and CPU time ($d_compute_{cpu}$) for FeatureMatch and ObjectDetect. It shows that $d_compute_{ts}$ is influenced by interference and $d_compute_{cpu}$ is robust.

the two applications run concurrently and induce interference. For ObjectDetect, $d_compute_{ts}$ has an evident fluctuation and deviates from $d_compute_{cpu}$, because it yields processor to FeatureMatch. The deviation is smaller for FeatureMatch because it rarely yields processor under the completely fair scheduling of Linux [115]. The result also shows that additional fluctuation caused by the interference arises in $d_compute_{cpu}$ of FeatureMatch when the two applications runs concurrently, which again addresses the need of reducing contention.

Heterogeneous workers: Because tasks of a client can run on a cluster of workers, if workers have different types of processors (e.g. number of cores, clock rate), the measured $d_compute$ is machine-dependent and can not be directly used for time series prediction. Even for an application with constant computing time, $d_compute$ may vary for different machines.

To handle this, we use a simple approach that assumes that more powerful workers speed up a task linearly. The speedup factor s_k is a constant for worker k and the following equation is satisfied for any two workers:

$$s_1 \cdot d_cputime_1 = s_2 \cdot d_cputime_2 \quad (5.14)$$

The predictor uses $s \cdot d_cputime$ as samples instead of $d_compute$. Then the prediction result is divided by s_k to obtain the predicted $d_cputime$, and Equation 5.13 is used to get the computing time (written as $d_compute'$).

5.5.2 Computing Time Prediction

Accurate prediction of computing time is significant for ATOMS. Under-estimation leads to failure in detecting resource contention, and over-estimation results in larger interval jitter and utilization loss. We use upper bound estimation for applications with low variability of computing time, and time series prediction for applications with high variability.

As introduced in Section 3.6.2, $s \cdot d_cputime$ is collected as samples for the prediction. Given that T_n is the last completed task of client C_i , instead of just predicting T_{n+1} , which is the next task to plan, ATOMS needs to predict T_{next} ($next > n$). $N_{predict} = next - n$ gives how far it predicts since the last sample, and is decided by the parameter d_future (Section 5.4.1) and the period of client, calculated by $\lceil d_future / d_period_i \rceil$.

Upper Bound Estimation

The first method estimates the upper bound of samples using TCP retransmission timeout estimation algorithm [76]. The method is similar to the estimation method for upstream network delays in Section 5.5.3. We denote the value to predict as y . The estimator keeps a smoothed

estimation y^s and a variation y^{var} . They are updated by a new sample y_i as:

$$y^{var} \leftarrow (1 - \beta) \cdot y^{var} + \beta \cdot |y^s - y_i|, \quad (5.15)$$

$$y^s \leftarrow (1 - \alpha) \cdot y^s + \alpha \cdot y_i. \quad (5.16)$$

The upper bound y^{up} is given by:

$$y^{up} = y^s + \kappa \cdot y^{var} \quad (5.17)$$

where α , β , and κ are parameters. This method outputs y^{up} as the prediction of $d_compute$ for T_{next} . This lightweight method is adequate for applications with low execution time variability, for example the ObjectDetect application. It tends to over-estimate for application with highly varying execution time because it uses upper bound as the prediction.

Time Series Linear Regression

Model: In the auto-regressive model for time series prediction problems, the value y_n at index n is assumed to be a weighted sum of previous samples in a moving window with size k . That is,

$$y_n = b + w_1 y_{n-1} + \dots + w_k y_{n-k} + \epsilon_n,$$

where y_{n-i} is the i th sample before the n th, w_i is the corresponding coefficient and ϵ_n is the noise term. We use this model to predict y_n . The input are the previous k samples measured by worker servers, y_{n-1}, y_{n-2}, \dots , and y_{n-k} .

We use the model to predict the $N_{predict}$ th sample after y_{n-1} using a recursive approach: to predict y_{i+1} ($i > n - 1$), the prediction on y_i is used as the last sample. This approach is flexible in predicting arbitrary future samples, however, as $N_{predict}$ increases, the accuracy degrades because prediction errors are accumulated.

Training: To train the weights, w_i , where $i \in \{1, 2, \dots, k\}$ and b , from a series of

measurements, we generate the following dataset with N samples,

$$\left[\begin{array}{c|cccc} y_k & y_{k-1} & y_{k-2} & \cdots & y_1 \\ y_{k+1} & y_k & y_{k-1} & \cdots & y_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ y_{k+N-1} & y_{k+N-2} & y_{k+N-3} & \cdots & y_N \end{array} \right],$$

where the first column indicates the labels and the rest rows are the features used to train a linear regression model.

The predictor keeps a model for each client which is trained either online or offline. In the online approach, previous samples are collected as mini-batches of training data. The model is re-trained on the latest mini-batches. The offline approach is simpler which trains the model once on a pre-collected series of samples. The offline approach has the advantage of being re-training free, however, the training data must be carefully selected for prediction accuracy.

Prediction Performance

We predict the computing time for the FaceDetect application as an example. The video input is from the Jiku dataset [133] with a rate of two frames per second to simulate an offloading period $500ms$. The ground truth of $d_compute$ is given in Figure 5.9 as the blue line. It gives the results for $N_{predict} = 1$ of upper bound estimation ($\alpha = 0.125$, $\beta = 0.125$, $\kappa = 1$) and online linear regression (mini-batch size is 100). We have several observations. First, the values in ground truth change very fast. Second, the upper bound method tends to over-estimate the values. Third, the regression model can fit the ground truth reasonably well.

The CDFs of prediction error ($d_compute' - d_compute$) for $N_{predict} = 1, 3, 5$ are given in Figure 5.10. It shows that comparing with linear regression, upper bound estimation tends to over-estimate when $N_{predict} = 1$. However, the performance of linear regression degrades for larger $N_{predict}$. We also present the CDFs of error using two offline models trained on two videos with entirely different context. Compared with the online method, the offline model 0 tends to

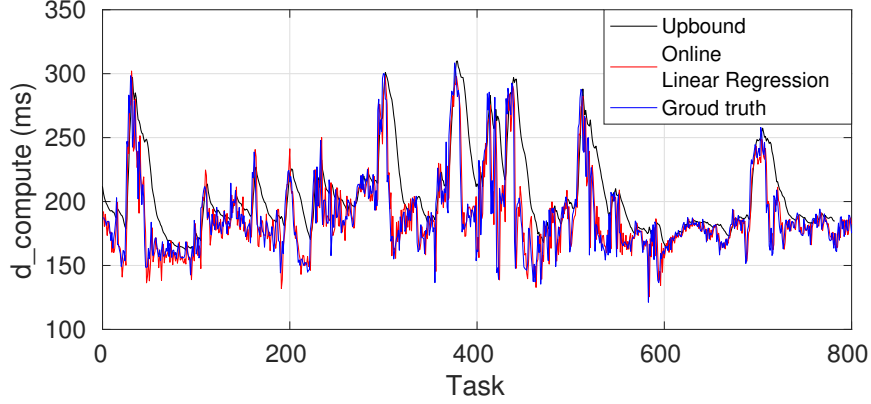


Figure 5.9. Prediction results of computing time for FaceDetect application, using upper bound estimator and time series linear regression.

over-estimate while the offline model 1 tends to underestimate. It shows that the performance of offline model is sensitive to the content of the video which generates the training data.

5.5.3 Upstream Latency Estimation

Our system is capable of estimating upstream network latency for task scheduling purposes. Because high spikes of d_{up} may be present, we estimate the lower and upper bounds (d_{up}^{low} , d_{up}^{up}) of upstream network latency d_{up} instead of the exact value. The TCP retransmission timeout estimator [76, 120] is adopted. It tracks a smoothed estimate of RTT (RTT^s) and computes the deviation (RTT^{var}) of measured RTT samples (RTT^m). The estimate of RTO is given by:

$$RTO_n = RTT_n^s + \kappa \cdot RTT_n^{var}, \quad (5.18)$$

where n is the number of sample. κ is a parameter. RTT_n^{var} and RTT_n^s are updated as follows with parameters α and β :

$$RTT_n^{var} = (1 - \beta) \cdot RTT_{n-1}^{var} + \beta \cdot |RTT_{n-1}^s - RTT_n^m|, \quad (5.19)$$

$$RTT_n^s = (1 - \alpha) \cdot RTT_{n-1}^s + \alpha \cdot RTT_n^m. \quad (5.20)$$

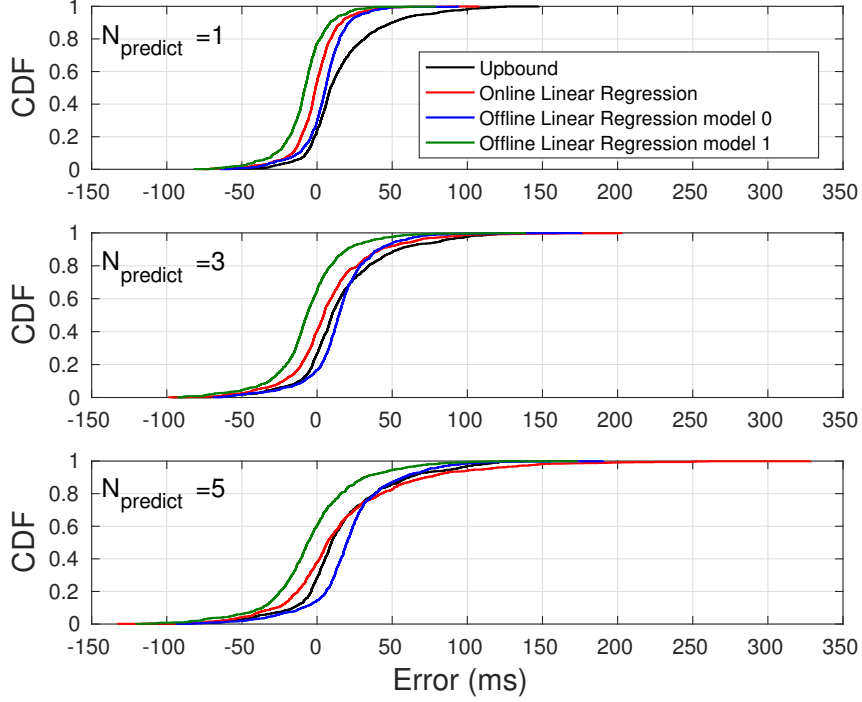


Figure 5.10. CDFs of prediction errors with $N_{predict} = 1, 3, 5$. We compare the upper bound estimator (Section 5.5.2), linear regression models trained online and offline (Section 5.5.2). It shows that prediction errors increase for larger values of $N_{predict}$.

Subtraction is used instead in Equation 5.18 for the lower bound. The estimator has a non-zero *error* when a new sample of d_{up} falls out of the range, calculated as its deviation from the nearest bound. The error is positive if d_{up} exceeds d_{up}^{up} , and negative if it is smaller than d_{up}^{low} . Because the uncertainty in estimation ($d_{up}^{up} - d_{up}^{low}$) is added into task reservation, larger uncertainty over-claims the reservation span, which causes higher jitter of interval and reduces the processor utilization of the system. Choosing the parameters is thus a trade-off between error and uncertainty.

We measure d_{up} of offloading 640×480 video frames through campus Wi-Fi. The size of frames varies from 21KB to 64KB. For the purpose of studying LTE networks as well, we use the Mahimahi tool [111] to obtain simulated latency of Verizon LTE. The CDFs of latency are plotted in Figure 5.11a. The maximal latency is 1.1s for Wi-Fi and 3.2s for LTE. We demonstrate the performance of estimation with the parameters $\alpha = 0.125$, $\beta = 0.125$, $\kappa = 1$. CDFs of

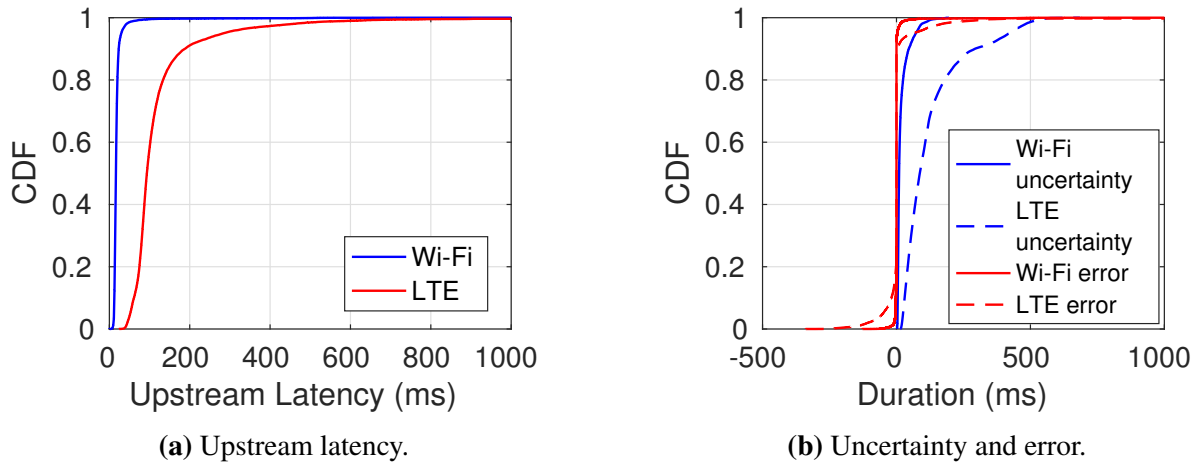


Figure 5.11. (a) CDFs of WiFi and LTE upstream network latencies, and (b) CDFs of uncertainties and errors of network latency estimation.

estimation error and uncertainty are plotted in Figure 5.11b. Results show that the uncertainty of Wi-Fi network is small, while it is very large for LTE (the maximal value is $2.6s$). We demonstrate how error and uncertainty influence the scheduling and offloading performance through experiments in Section 5.6.

5.5.4 Clock Synchronization

The client timestamps are synchronized to the server time in our system. We seek for a general solution of clock synchronization without the need to patch the OS of mobile client. The server is synchronized to the global time using NTP. Because time service now is ubiquitous on mobile devices, we require clients to be *coarsely* synchronized via NTP either to the global time or to the server.

NTP client [106] initiates synchronization periodically. In each synchronization round, it polls the NTP servers and collects responses. Each response R contains four timestamps taken respectively at the time of leaving the client (t_{org}), arriving at the server (t_{rec}), leaving the server (t_{xmt}) and arriving back at the client (t_{dst}). The clock offset δ_i and round trip delay RTT_i of the response R_i are computed from the timestamps. NTP protocol computes root distance r_i based on RTT_i , clock precision of the time server and clock drift compensation. The lower and

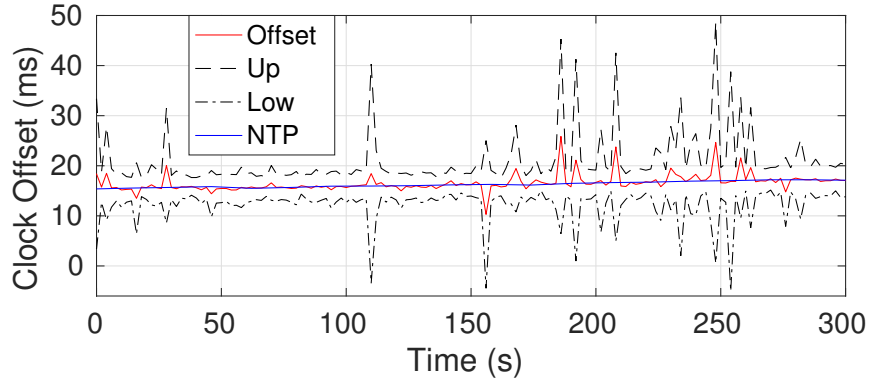


Figure 5.12. Clock synchronization results. NTP requests are sent every $2s$, the offset (red) with lower and upper bounds calculated from single NTP response are given. The result of NTP algorithm using $N_{ntp} = 8$ (blue) is invulnerable to RTT fluctuation.

upper bounds of clock offset are obtained as $[\delta_i^{low}, \delta_i^{up}] = [\delta_i - r_i, \delta_i + r_i]$. After collecting all responses, NTP protocol applies four algorithms to compute the final offset δ^f : clock filter, selection, clustering and combining. The selection algorithm is based on Marzullo’s algorithm [103]. It finds a non-empty intersection interval $[\delta^{low}, \delta^{up}]$ of offset bounds, which contains all offsets δ_i of the correct responses. The selection algorithm runs iteratively to exclude false responses until the intersection interval is found. The synchronization trial fails if the rest correct responses are not the majority. The combining algorithm outputs the final clock offset δ^f using the correct responses. Details of NTP synchronization algorithm can be found in [106].

We use the following approach for *fine* clock synchronization. Client sends out a NTP synchronization request to the master each time it receives offloading result to avoid the wake-up delay [129]. Although the client can calculate clock offset using just each single NTP response, in order to eliminate the influence of packet delay spikes, it buffers N_{ntp} responses and runs the NTP algorithm [106]. It applies clock filter, selection, clustering and combining algorithms to N_{ntp} responses and finally outputs a robust estimate on clock offset. ATOMS uses the estimated clock offset to synchronize timestamps exchanged between clients and the master.

In Figure 5.12, we demonstrate the clock synchronization on one client (C_0 , the detailed setup is described in Section 5.6.2). The client gets an NTP response per offloading cycle ($2s$).

The values of offset, lower and upper bounds ($offset \pm RTT/2$) are plotted as the red and two dashed black lines respectively. Due to spikes in RTT of NTP packets, the offset of single response has frequent fluctuation and is thus less reliable. The blue line gives the offset obtained by the NTP algorithm using batches of $N_{sample} = 8$ responses. It proves that our approach is able to get rid of RTT uncertainty and gets a robust estimate on clock offset.

5.6 Evaluation

We evaluate ATOMS through tests on real-world applications, that evaluate performance by ATOMS as well as trace-driven emulations to explore its performance while managing more worker machines and serving more clients.

5.6.1 Test Setup

FCFS scheduling is used as the baseline for comparison. It simply dispatches an incoming task to the first available computing engine, without the planning phase. A task is buffered in a global FIFO queue if there is currently no free engine. We compare E2E delay and offloading interval between FCFS and ATOMS. In ATOMS, the prediction for FaceDetect and FeatureMatch uses offline linear regression, and the upper bound estimator is used for ObjectDetect. The network latency estimation uses parameters $\alpha = 0.125$, $\beta = 0.125$ and $\kappa = 1$. We set d_inform (discussed in Section 5.4.1) to 300ms for all clients in tests.

We simulate the camera feed using the approach as [37] to conduct reproducible experiments. Each client selects which frame from a video stream to offload based on the current time and the frame timestamp. We use three public video datasets as the camera input: (1) the Jiku mobile video dataset [133] is used by FaceDetect. A video clip is used for each client; (2) Piecewise-Planar StereoScan [126] provides videos from a binocular camera to feed the Feature-Match application. A pair of left and right images is shipped per task; (3) we use multi-camera pedestrians videos [50] for ObjectDetect, where the detection captures a crowd of pedestrians. We resize the frames to 640×480 in all the tests. Each test runs for 5 minutes.

We evaluate the performance of ObjectDetect application using a local setup. We use RaspberryPi board as client. They connect to the ATOMS master, which runs on a 4-core (Intel i5-6600, 3.3GHz) desktop in the same building through public campus WiFi hotspot. There is one worker machine (8-core, Intel i7-4790, 3.6GHz) hosting a computing engine of ObjectDetect.

The other evaluations are conducted on AWS EC2. The master runs on a c4.xlarge instance (4 vCPUs and 7.5GB memory). Each worker machine is a c4.2xlarge instance (8 vCPUs and 15GB memory). We emulate all clients on a single c4.2xlarge instance because the processor and memory overheads of client are very low. Pre-collected network upstream latencies (as described in Section 5.5.3) are replayed at each client to emulate the wireless network. Each offloaded task is delayed for a latency before being sent out.

5.6.2 Application Performance

We first evaluate the improvement on the application performance of ObjectDetect. Four RaspberryPi boards are used as clients (C_0, C_1, C_2, C_3) in the test. Their offloading periods are 2s, 3s, 4s and 5s respectively. Clients start offloading with a random offset within 5s. The planning length of ATOMS is $d_{future} = 6s$.

Offloading metrics: The offloading performance is shown in Figure 5.13. The delay plot (bottom) shows that when FCFS is used, multi-tenancy interference aggravates E2E delay of all the clients. ATOMS effectively mitigates the interference and significantly improves the delay performance. The interval plot (top) shows that offloading intervals become non-constant, caused by task coordination. C_0 has the smallest period and is more likely to be interfered by others. It benefits the most from delay improvement: the 50% percentile is reduced from 872ms to 443ms, and the 90% percentile is reduced from 1302ms to 466ms. Due to more frequent adjustment, C_0 has the largest jitter that the interval falls in the range from 1.6s to 2.4s.

Trackability: Longer delay causes trackability loss in object detection. To demonstrate this, we use intersection over union (IOU) [43] as the metric of successful detection and localization. The precision is the ratio of the number of objects accurately detected ($IOU > 50\%$) using

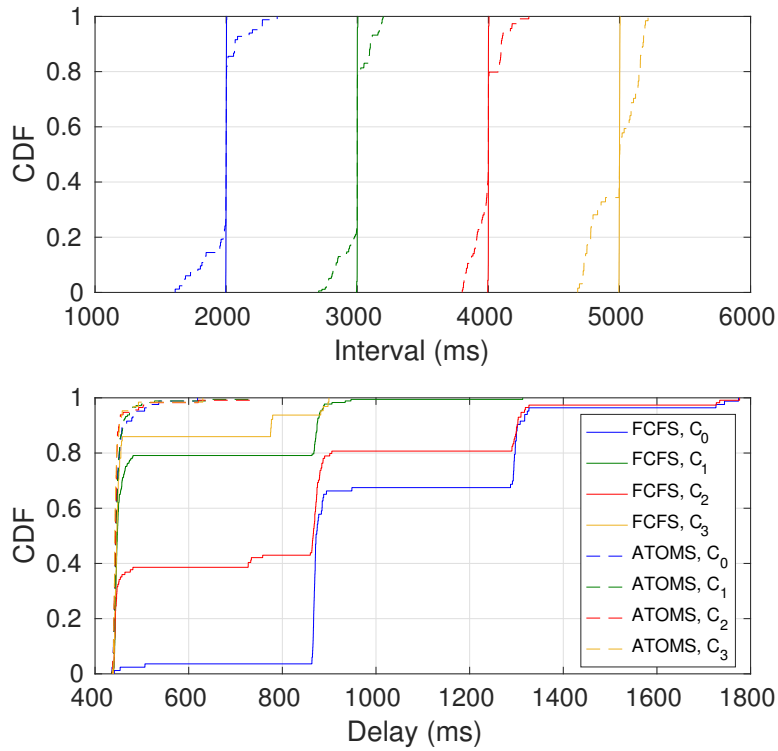


Figure 5.13. Compare FCFS and ATOMS in offloading interval (top) and E2E delay (bottom). The average CPU utilization is 50% for both FCFS and ATOMS.

offloading over the detected objects by the scheme itself [37]. The video inputs of four clients are taken by cameras under different angles filming the same crowd of pedestrians. Trackability varies for different videos due to the difference in pedestrian moving speeds. The result in Figure 5.14 shows that ATOMS improves precision for all clients, where C_0 gets the largest precision improvement (over 20%), because it suffers from the worst delay without ATOMS.

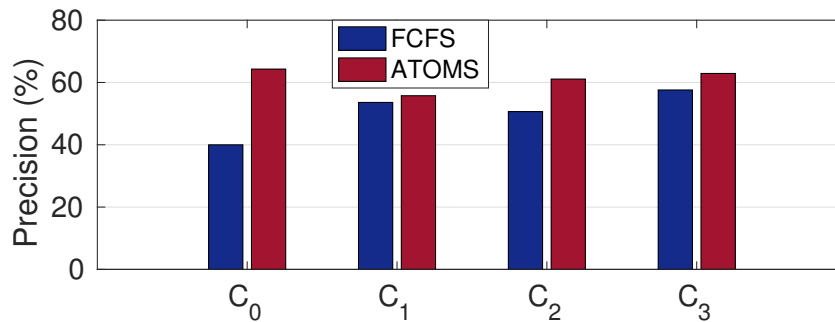
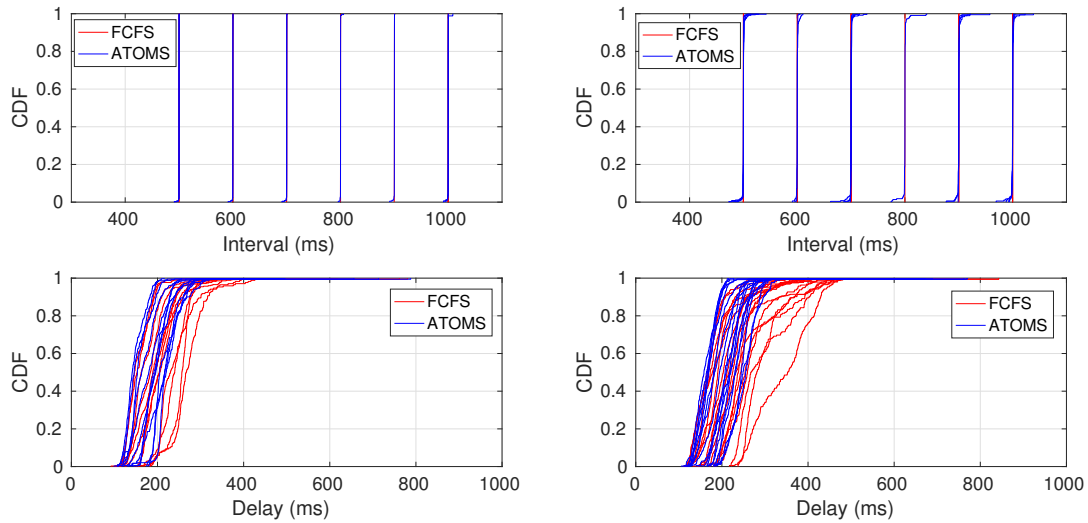
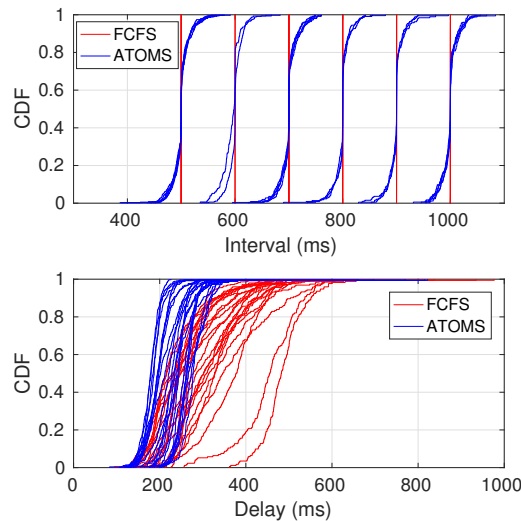


Figure 5.14. Precision of pedestrian detecting and tracking using FCFS and ATOMS.



(a) 12 clients. The average CPU utilization is 38% for FCFS and 36% for ATOMS. (b) 18 clients. The average CPU utilization is 58% for FCFS and 55% for ATOMS.



(c) 24 clients. The average CPU utilization is 83% for FCFS and 81% for ATOMS.

Figure 5.15. Offloading performance of FCFS and ATOMS running FaceDetect using WiFi. CDF of each client is plotted.

5.6.3 Serving More Clients

We explore the performance of ATOMS when it serves increasingly more clients. In the AWS testbed, we use 12 to 24 clients running FaceDetect, with periods ranging from 0.5s to 1s. $d_{future} = 2s$ is used in planning. We use one worker machine (8 vCPUs) hosting 8 FaceDetect

engines. A fully partitioned scheme is used that the planner has a reservation queue for each engine with $Q_{cpu} = 1$. We first use WiFi network, as shown in Figure 5.15, the interference becomes more intensive as the number of clients gets larger, and FCFS suffers from increasingly dragged E2E delay. ATOMS is able to maintain low E2E delay even the total CPU utilization is over 80%, use the 24 clients case as example, 90% percentile tail E2E delay is reduced by 34% in average for all clients, and the maximum reduction is 49%. The interval plots (top) show that jitter of offloading interval increases in ATOMS which is caused by task coordination.

In order to investigate how ATOMS performs under larger network latency and variation, we run the test with 24 clients using LTE latency data. As discussed in Section 5.5.3, to tolerate the estimation uncertainty of upstream latency, which has a 90% percentile as large as 290ms for LTE, the reservation that includes task arriving time uncertainty becomes large. In this case, the consequence is that the total reservation exceeds the total processor capability. As shown in Figure 5.16, the planner has to postpone all reservations to remove contention. All clients thus have severely dragged interval as the blue lines. We propose to disable the uncertainty estimation ($\kappa = 0$) for LTE to serve more clients without interval degradation. The result shows that without network uncertainty, the offloading interval can be maintained (green lines in Figure 5.16). We show the 50% and 90% percentiles of E2E delay in Figure 5.17, for each client, using FCFS (red), ATOMS with network uncertainty (blue) and ATOMS without network uncertainty (green). It shows that without network uncertainty, most clients have larger delays, but it still presents reasonable improvement: 90% percentile of delay is decreased by 22% in average and by 28% as the maximum among all clients.

5.6.4 Shared Worker

Contention mitigation is more complex for shared workers. To conduct the evaluations, we set up 4 ObjectDetect clients with periods 2s and 3s, and 16 FeatureMatch clients with periods 2s, 2.5s, 3s and 3.5s. $d_{future} = 6s$ is used in the planner. We use 4 shared workers (c4.2xlarge instance), and each hosts 4 FeatureMatch engines and 1 ObjectDetect engine.

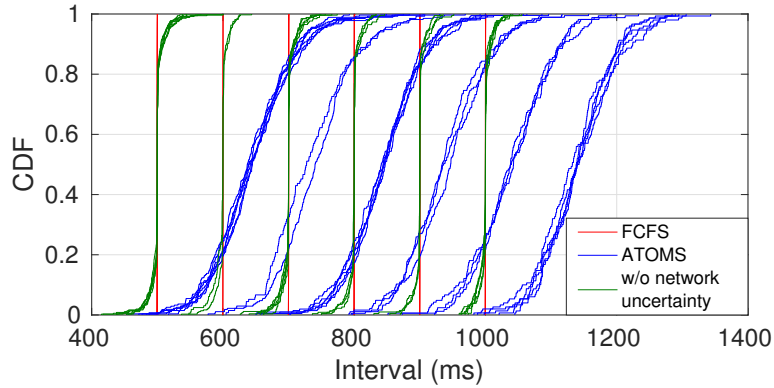


Figure 5.16. Offloading interval running FaceDetect using LTE. The average CPU utilization is 83% for FCFS, 59% for ATOMS with network uncertainty, and 81% for ATOMS without network uncertainty.

Planning schemes: We compare three schemes of planning: (1) a global reservation queue is used for 4 workers with $Q_{cpu} = 32$; (2) 4 reservation queues ($Q_{cpu} = 8$) are used and Best-Fit is used to select queue; (3) 4 queues with Worst-Fit selection. We adopt the load-aware scheduling with Worst-Fit worker selection. The CDFs of interval (top) and E2E delay (bottom) of all clients are given in Figure 5.18, separated by the application. It shows that Worst-Fit more aggressively adjusts tasks thus leads to the largest jitter of interval. The reason is that when Worst-Fit is used, FeatureMatch tasks which have low parallelism tend to be evenly allocated to all reservation queues. It becomes more like to cause resource contention when ObjectDetect is placed, which demands all 8 cores. Its advantage is the improved delay performance. Shown by the delay plots in Figure 5.18, Worst-Fit (black) performs evidently better for the 4 ObjectDetect clients comparing with Global (red) and Best-Fit (blue), especially the worst delay (100% percentile) of the 4 clients that is 786ms for Worst-Fit, 1111ms for Best-Fit and 1136ms for global. The delay performance of FeatureMatch is similar for the three schemes.

Scheduling schemes: Figure 5.19 shows the E2E delay using different scheduling schemes: (1) a simple FCFS scheduler which selects the first available engine; (2) the load-aware scheduling with Best-Fit worker usage selection; (3) the load-aware scheduling with Worst-Fit selection. The planning uses 4 reservation queues with Worst-Fit selection. For the

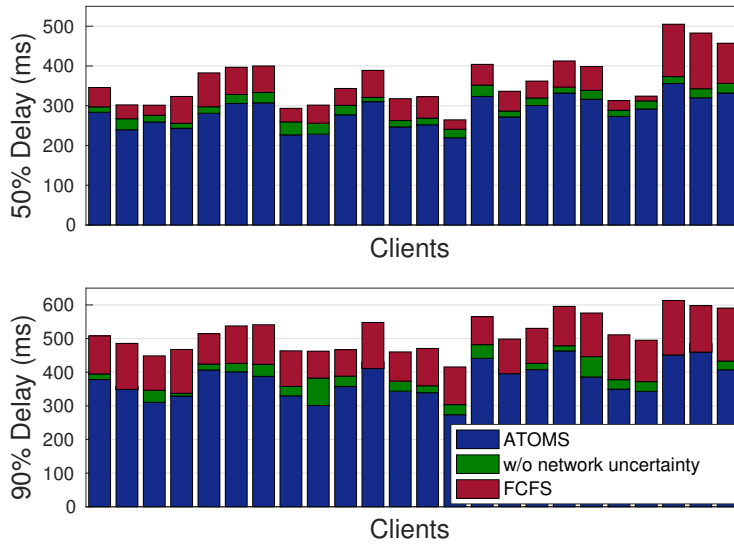


Figure 5.17. 50% and 90% percentiles of E2E delay of all clients running FaceDetect using LTE.

simple scheduling, ObjectDetect tasks are more likely to be influenced by contention, because it tries to run in parallel on all 8 cores. FeatureMatch requires 2 cores at most and can get enough processors more easily. Best-Fit performs the best for ObjectDetect, whereas it degrades dramatically for FeatureMatch clients. The reason is that the scheduler tries to pack incoming tasks as tightly as possible on workers. As the consequence, it leaves enough space to schedule highly parallel ObjectDetect tasks. However, due to the error of computing time prediction and network estimation, higher possibility of contention occurs for tightly scheduled FeatureMatch tasks. The Worst-Fit method has the best performance for FeatureMatch tasks and still maintain reasonably low delay for ObjectDetect. Therefore it is the most suitable approach in this case.

Figure 5.20 compares the 50% and 90% E2E delay of all clients between FCFS and ATOMS (Worst-Fit scheduling). In average ATOMS reduces the 90% percentile E2E delay by 49% for ObjectDetect clients, and by 20% for FeatureMatch clients.

5.6.5 Service Level Agreement

We now consider the use and limitation of SLAs at the end. One 8-core worker serves 4 ObjectDetect clients with period 2s to 5s. As shown in Figure 5.21, the red line gives the

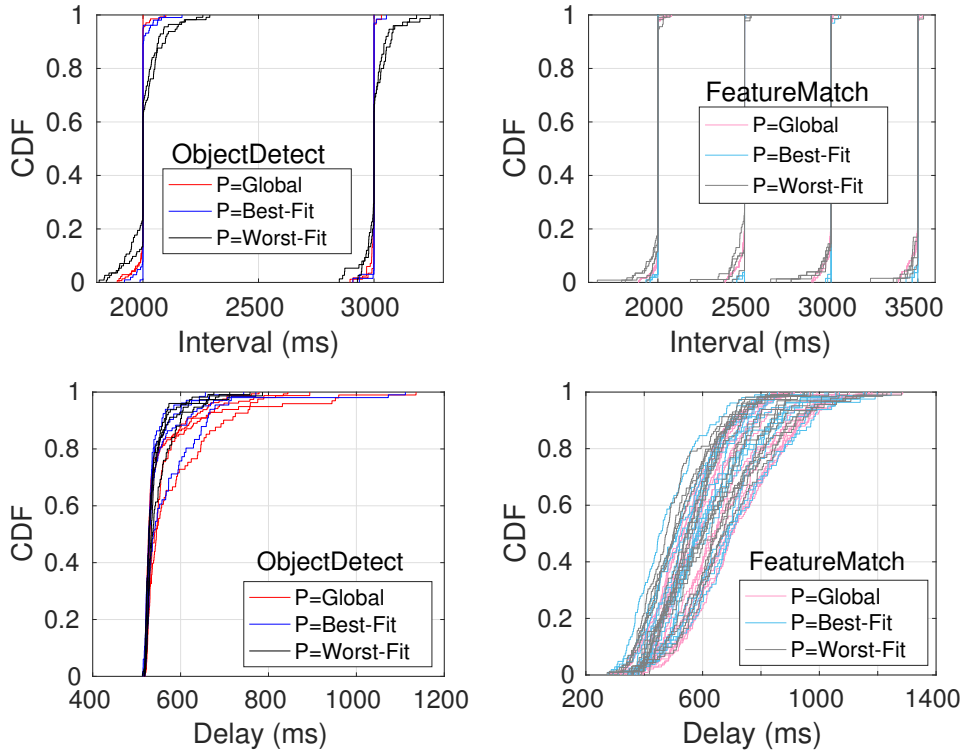


Figure 5.18. Offloading interval and E2E delay of 4 ObjectDetect and 16 FeatureMatch clients, using different planning schemes. The average CPU utilization is 36% in all cases.

baseline where no SLA is given. Then client C_0 specifies the following SLAs: $sla^{10\%} > 1.9s$ and $sla^{90\%} < 2.1s$. The SLAs are plotted as back circles in Figure 5.21a. The blue lines give the CDFs of interval with the SLAs. The planning algorithm will not adjust C_0 when the SLAs are violated, because it is the only client with non-zero cost. C_0 thus has less jitter of interval. The other clients are influenced by the SLAs of C_0 , for example, C_1 and C_3 experience larger interval.

Because SLAs represent client's priority rather than guaranteed performance, when more clients have tough SLAs, it becomes harder to achieve the targeted goal. In Figure 5.21b C_2 have SLAs $sla^{10\%} > 3.9s$ and $sla^{90\%} < 4.1s$. Due to the interference between C_0 and C_2 , $sla^{90\%} < 4.1s$ is not satisfied for C_2 , and the tail interval jitters of C_0 evidently increase. SLAs should be use to prioritize clients, however, guaranteeing offloading performance of all clients leverages on admission control in addition, which lies in the future work.

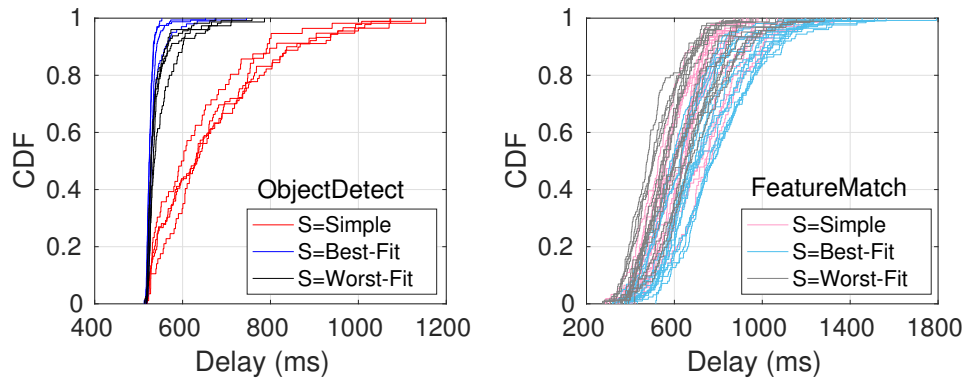


Figure 5.19. E2E delay of ObjectDetect and FeatureMatch using different scheduling schemes. The average CPU utilization is 36% in all cases.

5.7 Summary

We present the ATOMS scheduler in this chapter. ATOMS predicts the time slots that should be reserved for future offloaded tasks, and coordinates them to mitigate processor contention on servers. It selects the best server machine to run each arriving task to minimize contention, based on the real-time workload on each machine. The realization of ATOMS is achieved by key system designs in computing time prediction, network latency estimation, distributed processor resource management, and client-server clock synchronization. The experiments prove the effectiveness of ATOMS in improving the E2E delay for applications with various parallelisms and computing time variabilities.

This chapter, in part, is a reprint of the material as it appears in Zhou Fang, Mulong Luo, Tong Yu, Ole J. Mengshoel, Mani B. Srivastava, and Rajesh K. Gupta. “Mitigating Multi-Tenant Interference in Continuous Mobile Offloading”, in Proceedings of the 2018 International Conference on Cloud Computing (CLOUD’18), 2018. The dissertation author was the primary investigator and author of this paper.

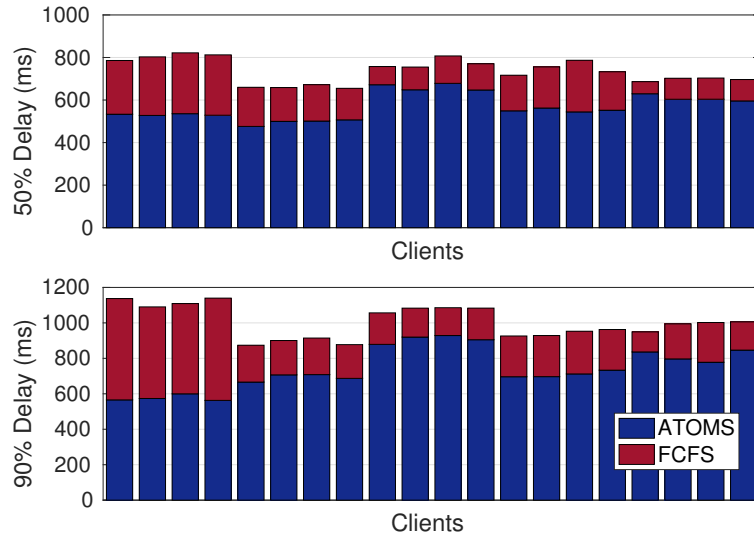


Figure 5.20. 50% and 90% percentiles of E2E delay of ObjectDetect (bar 1 to 4) and FeatureMatch (bar 5 to 20) clients. The average CPU utilization is 40% for FCFS and 36% for ATOMS.

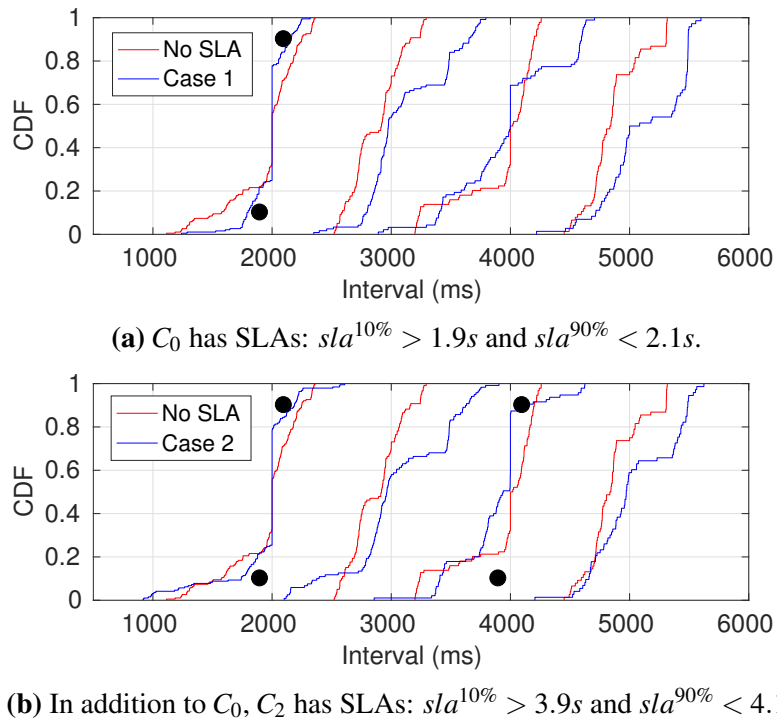


Figure 5.21. Offloading interval of 4 ObjectDetect clients. The average CPU utilization is 59%.

Chapter 6

Data Flow-Based Mobile Offloading APIs

This chapter presents high-level APIs and the underlying system implementation to program a mobile application using a data flow model, and to partition it across different tiers of resources for trade-offs between resource usage and offloading performance.

6.1 Introduction

The DeepQuery APIs presented in Chapter 3 generate a query for every new vision task, *e.g.*, object detection in a new frame. For a vision application that processes a continuous video stream, the offloaded computation is recurring across frames. Such an application can be programmed using a data flow model: An application is composed of sources of stream data and processing modules processing the data, connected with each other as a DAG. This programming paradigm provides the benefits of simple application code, high scalability, and ease for fault-tolerance. It has already been widely adopted in big data analytics frameworks [45, 9, 159, 145, 26].

This chapter presents our system design and implementation for programming an offloaded vision application using a data flow model into a framework named *DeepVision*. It supports partitioning an application between the mobile and the server, which has been explored in previous works to optimize on-board power consumption, network bandwidth usage, application performance, and E2E delay [112, 41, 121, 37, 82]. To implement the capability to

partition an application in DeepVision, data processing modules can be placed on different tiers of computing resources. Compared to data processing in a cluster [159, 145, 26], the mobile computing scenario brings about new design considerations: (1) The data stream between the mobile and the server should be minimized to save on-board power consumption and network bandwidth; (2) The compute overhead of processing module must consider heterogeneity of the resources, *e.g.*, resource efficient algorithms should be used on the mobile.

Similar to DeepQuery (see Section 6.3.3), we target emerging mobile applications that leverage multiple vision tasks, in particular DNNs, for deeper analysis into live video data. Figure 6.1 gives an example of a video analytics application. At the object level, a video object detection task detects objects using DNN models. Objects are tracked across frames and the traces of unique objects are stored as *tracklets*. The key frame selection task selects significant frames heuristically. It uses a greedy search method to select a minimal subset of frames that contain all tracklets. The CNN feature extraction and video description tasks together generate a sentence that describes the latest video clip. A clip contains 512 frames (20.48s for 25fps) in our example. The application can be built in DeepVision as a DAG of data flows, where each node in the DAG is a microservice processing the above vision tasks.

This chapter presents the design and implementation of DeepVision. DeepVision builds a vision application in the form of a DAG of *queries* that process data streams. A query is forwarded to a microservice (denoted as *service*) for processing, which can be placed on a mobile device, cloudlet, or cloud servers. A service encapsulates an algorithmic module and exposes a microservice interface. It runs in processes on mobile devices and in Docker containers [5] on servers. DeepVision automates the deployment of microservices and lets application developers focus on algorithms. We use the application in Figure 6.1 to motivate the design of DeepVision with evaluations on a mobile platform (NVIDIA Jetson TX2 board).

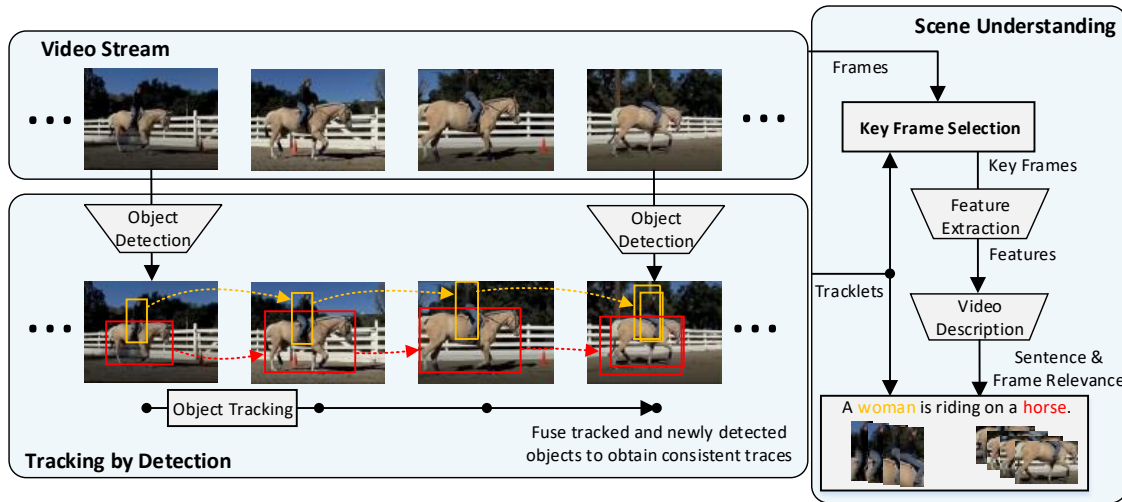


Figure 6.1. An intelligent video analytics application that provides both high-level descriptions and low level object traces. It detects and tracks objects, and maintains a continuous trace for each unique object (tracklet). At a higher level, it generates a description of the latest stream and highlights the tracklets related to the description.

6.2 Background

The data flow model has been widely adopted in big data frameworks [159, 145, 26]. It constructs a distributed data processing pipeline as a DAG of transforms on data streams. Compared to the MapReduce model [45, 9], it is more powerful in expressing complex pipelines and providing real-time processing capability. Tremendous system researches on big data frameworks have deeply explored the data flow model in aspects of scalability, fault-tolerance, and latency reduction [159, 145, 26]. For example, Spark [159] introduced resilient distributed datasets (RDDs), an abstraction of data partitioned across a set of machines, which can be rebuilt to provide fault-tolerance capabilities. Spark keeps data in memory to provide low processing latency for iterative analytics applications. Storm [145] is a distributed data processing framework for real-time workloads. It supports the “at least once” and “at most once” semantics in data processing, which is enabled by reporting the processing status of data to a master server.

As we have introduced in Section 3.2, processing video data in the cloud, which is a

special class of big data workload, has been studied in previous works [100, 160, 72]. For example, Optasia [100] builds vision application using the SCOPE [35] dataflow engine, with design patterns including extractors, processors, reducers, and combiners. VideoStorm [160] adopts a master-slave architecture to processing queries on worker machines. The profiler and scheduler components on the master server is responsible for determining query configuration and placement. VideoEdge [72] further explores the query selection issues for a hierarchical cluster including cameras, private clusters, and public clouds.

Compared to the previous works on processing data in clusters [159, 145, 26, 100, 160, 72], this work adopts the data flow model to design a mobile offloading framework, which partitions an application between the mobile and the servers. The considerations in resource efficiency on mobile device and wireless network usage have been addressed in our implementation.

6.3 Framework Design

We first describe the APIs to build a video analytics application, and then the design of our underlying distributed systems.

6.3.1 Application Programming Interfaces (APIs)

We begin by describing the collection of APIs to create a DAG of queries. A DAG is defined by the class `QueryGraph`. Figure 6.2 illustrates the query graph for the application (Figure 6.1). A query is an algorithmic module that processes the input data and returns the result. Referring to our example, a query `Detection` that processes a batch of video frames (`frames`) returns a batch of lists of bounding boxes (`dets`). Every list of bounding boxes comprises the detected objects in a frame. The input and output data (`frames`, `dets`) have type `Data`. It is a class that represents the values to be evaluated at runtime. Queries are connected via `Data`. For instance, a downstream query `Tracking` takes `frames` and `dets` as its inputs.

A graph also contains several utilities for data transform. For example, `Downsampling` downsamples the frames by 16 times for `Detection`. `Buffer` collects a batch of 512 frames and

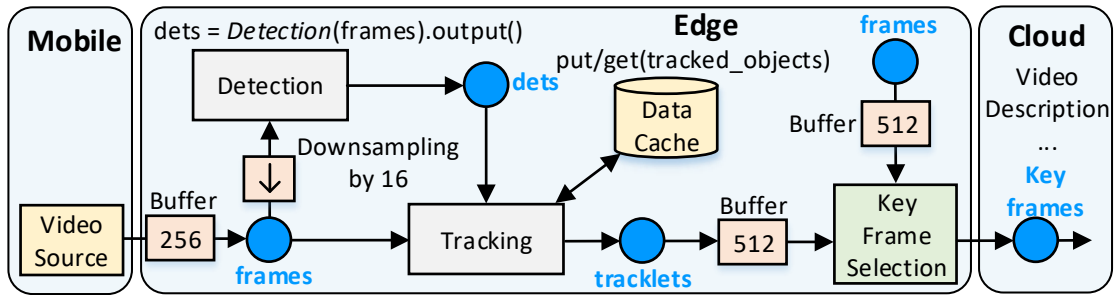


Figure 6.2. A sample query graph. The system comprises three resources: model, cloudlet (edge) and cloud. The mobile sends video frames to the edge. The first buffer collects a batch of 256 frames, then it runs detection and tracking tasks on this batch. The tracking task runs on every frame, and the detection task runs on every 16 frames. The second buffer collects tracking results from 512 frames, then key frames among the 512 frames are selected based on the tracked objects. The key frames are sent to the cloud for video description related tasks.

forwards it to the query KeyFrameSelect. To facilitate dynamic computation partitioning and scaling-out at runtime, queries are stateless and can be executed on one of several servers. The stateful variables are explicitly stored in data caches, such as the tracked objects from the output of Tracking. The graph is executed by providing the input values and the names of desired outputs. A compute resource is represented by the class Executor in the APIs, parameterized by the resource machine type (*e.g.*, mobile, cloudlet, cloud) and other metadata (*e.g.*, host IP and port of the executor machine). To allocate a query to be processed at a resource, an executor that represents the resource should be assigned to the query.

6.3.2 Queries and Services

A service represents a group of workers with the same functionality that process corresponding queries. The workers run as individual processes on mobile devices and are deployed as Docker containers on servers. The framework provides the query classes for common vision tasks, such as Detection and Tracking. To define a new query class, the user needs to define a remote procedure call (RPC) stub for the query to connect to the service. Services hosted on mobile devices or running lightweight tasks can be created as *local services*, which run

on local processes and are not connected through RPC stubs. We assume that in production deployment, cloudlet and cloud clusters provide ready-to-use services for common tasks. For example, as the core primitives for many mobile applications, object recognition and detection services are expected to be publicly available for clients using the system. The query to a public service contains the service type and the parameters for data processing (*e.g.*, DNN model configurations). At the running time, the system routes queries to the correct services based on the service types and parameters.

If the service for a query is not currently available, the query has to provide a method to launch a new service on servers. For this purpose, the query contains a serializable Transform class to process data for simple algorithms. And for complex algorithms, the queries should contain URLs or Dockerfiles of the container images to create containers that run and expose the service.

6.3.3 Distributed Systems

The underlying distributed system to support the described APIs is illustrated in Figure 6.3. At its core, a compute resource is abstracted as an executor implemented using a master-workers architecture. DeepVision’s workflow begins with an initialization phase: the graph QueryGraph is partitioned into subgraphs, each contains a set of queries and data cache operations (described in Section 6.3.1), to be executed on several executors.

Our design combines two general approaches for distributed data processing: (1) *distributed*: data is processed and delivered to the next processing module according to the data dependency relationship given by a DAG, *e.g.*, stream processing systems like Storm [145]; and (2) *centralized*: a master server sends workloads to multiple worker servers and aggregates the results back for post processing (*e.g.*, web servers).

We adopt (1) to implement the data flow between subgraphs. Compared with using (2), the video stream captured from a mobile device is only transferred to a minimized number of servers, instead of sending stream from the mobile device (data source) to multiple servers

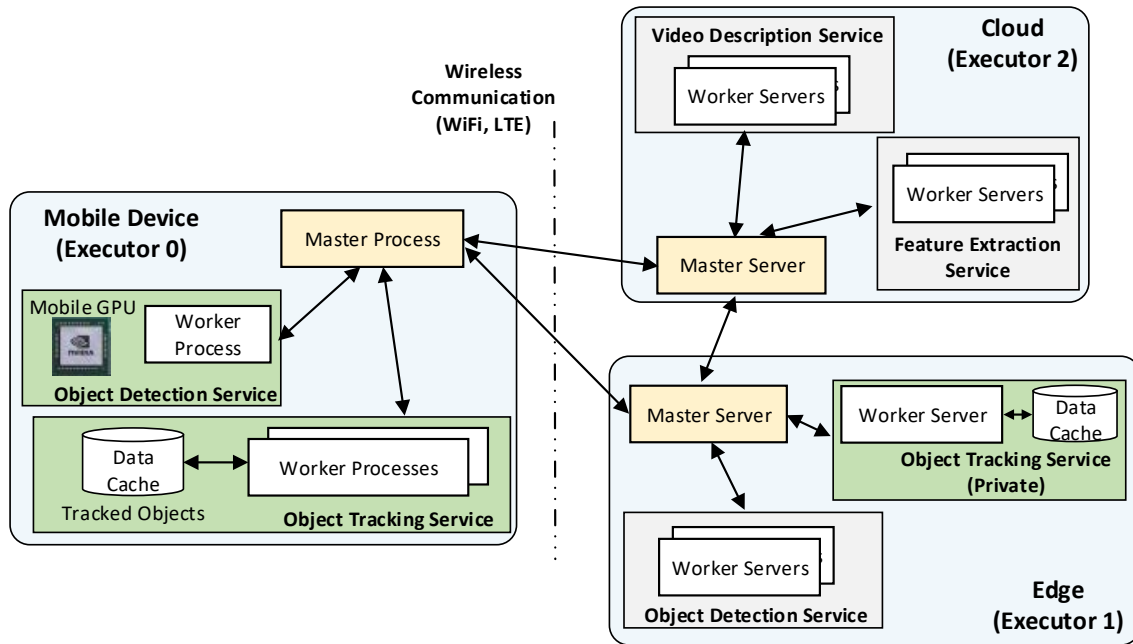


Figure 6.3. The architecture of DeepVision. An executor represents a compute resource that provides several services. Services run as one or more workers, coordinated by a master. The services can be either shared by several clients or be private to one client (marked as green).

and aggregating results at the mobile side. The device receives back the results only when all computations have been completed: the overheads of wireless communication are thus minimized. An executor computes a subgraph using (2), because the communication cost inner an executor (*e.g.*, inter processes, inner cluster) is low. Inside an executor, the master is responsible for sending all queries to services and aggregating the results. The design of executor gets simplified using a master-worker type architecture, due to the advantages in flexible service deployment, scalability and the supports for multi-tenancy.

6.4 Evaluation

We evaluated DeepVision using the application in Figure 6.1. We demonstrate that the framework effectively facilitates the process to build and deploy distributed vision applications. Changes in deployment, *e.g.*, data batching and computation allocation, can be easily made for

resource and performance trade-offs.

6.4.1 Experimental Setup

The evaluation is conducted on the NVIDIA Jetson TX2 device, one of today’s state-of-the-art mobile platforms. It is equipped with a NVIDIA Pascal GPU (256 CUDA cores) and 8GB memories shared by the GPU and CPUs. We refer to [27] for more details of the TX2 device. We use the on-board INA226 power monitor to measure the real-time power usage of GPU, CPU, WiFi and the entire device.

The device can offload workloads to several local servers (as an edge cluster), each equipped with a GeForce GTX 1080 GPU. The device connects to the servers through WiFi networks. The services are implemented using OpenCV and Tensorflow. We use the Single Shot Multibox Detector (SSD) model [97] for object detection. The SSD model uses MobileNet [70], a lightweight model designed for mobile devices, for feature extraction. A soft attention based encoder-decoder sentence generation model is adopted for video description [156], with input CNN features obtained from ResNet-50 [66]. We use Lucas Kanade algorithm [101] for object tracking and Hungarian algorithm [87] for matching tracked and newly detected objects.

Batch size is an important knob for DNN inference tasks. Although inference without batching gives the lowest delay, a larger batch size can lead to a higher throughput. We measured the inference delays for the detection task (SSD-MobileNet) and the CNN feature extractor (ResNet-50) on the mobile device and the servers, given in Table 6.1. On TX2, it achieves a $\times 5$ higher throughput for detection and $\times 2.3$ for CNN feature extraction using a batch size 16. It implies that although the input video comes in as a continuous stream, a proper batch size can improve the throughput and thus increase the resource efficiency. It is especially beneficial for resource constrained mobile devices. We also measured the delays of the video description model, which is 1135ms for TX2 and 103ms for GTX 1080 in average.

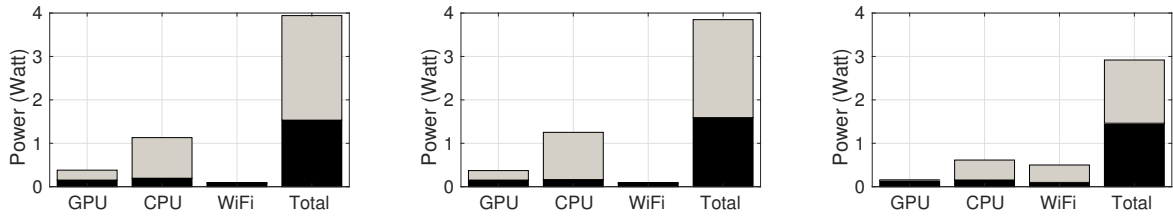
Table 6.1. DNN inference delays of the SSD-MobileNet and ResNet-50 models, measured on a TX2 board and a GTX 1080 GPU (ms).

Model	Device	Batch 1	2	4	8	16	32	64
SSD-MobileNet	Jetson TX2	251	526	943	1560	3248	-	-
	GTX 1080	59	112	216	433	858	1740	3434
ResNet-50	Jetson TX2	114	151	311	548	809	-	-
	GTX 1080	7	10	15	26	47	94	178

6.4.2 Examining Onboard Processing and Offloading

We first execute the application on-board and measure the system overheads. Because the video description runs for every 512 frames (20.48s for frame rate 25fps) in the application specification, the goal is to complete all computations within that time slot. We run object detection at a period of 16 frames (0.64s), and track objects in every frame. Observing the throughput improvement that comes with batching, the object detection task runs on a batch of 16 frames. Under this setting, the tracking and detection takes 7.58s for 512 frames. As a comparison, using batch size 8 in the detection results in a delay of 7.77s and it is 8.48s for batch size 4. It shows that a larger batch size reduces the total processing time, although sacrificing the time it takes to deliver tracking results to users. Using a batch size 16 in the detection, the total processing delay 9.30s is under the time budget.

We consider two ways to partition the workload of the application at different resources. The first one offloads feature extraction and video description tasks. It only sends key frames obtained from `KeyFrameSelection` to the server. The communication overhead is thus fairly low, comparing with sending the whole video stream. It is useful when there is a stringent delay requirement on object tracking and detection, because without the on-board loads of video description tasks, we can use smaller batch sizes. The other way is to offload all tasks to the server. In this way the delays and power consumptions of wireless communication become dominant, because all frames for detecting and tracking objects are sent to the server. The benefit is that more powerful DNN models (*e.g.*, Mask RCNN [65] for object detection) can be adopted instead of lightweight DNN models, when the models run at the server side. We



(a) Run all tasks on-board without offloading. (b) Offload feature extraction and video description. (c) Offload and run all tasks on the sever.

Figure 6.4. Power consumption comparison of different task placements. We observe that offloading all tasks to servers effectively reduces total on-board power consumption.

compare the power usages of on-board processing and offloading approaches in Figure 6.4. It shows that offloading all tasks to servers lessens the total power consumption to 76% of the on-board computing power usage.

6.5 Summary

We present the design of DeepVision, a framework that runs video analytics applications on mobile platforms backed by resource-rich servers. An application is built as a DAG of vision tasks processed by corresponding algorithmic modules. DeepVision provides utilities to manipulate data flows and route the data between algorithmic modules, which facilitate system scaling-out and partitioning an application onto different compute resources. The use of DeepVision in programming and running vision applications was demonstrated through evaluations on a NVIDIA TX2 device.

Chapter 7

DNN Adaptive Batching for Accuracy and Delay Trade-offs

In this chapter, we study the adaptive batching problem for DNN inference queries on servers that require sub-second delay. Our approach dynamically selects the model variant and batch size for every DNN query to maximize the average performance of recent queries in a time window. Different from Chapter 4 where we consider placing several DNN models for different vision tasks on shared GPUs on cloudlet servers, we consider general DNN serving frameworks [63, 40, 31] where each server handles queries for the same task, which work for both cloudlet and cloud systems.

7.1 Introduction

Accuracy for cloud computing workloads can be measured in different forms, such as quality of web responses [129], errors in big data analytics query [25], data staleness [144], *etc.* For DNN inference tasks, for example, accuracy is measured by errors of predicted labels for object recognition tasks, and localization errors are considered in addition for detection tasks. A higher inference accuracy is usually attained by adopting more complex network architectures, thus it comes with a longer processing delay. In order to maximize QoS of query responses, the scheduler of a DNN serving system should handle the *model selection* problem to consider the two contradictory objectives: *accuracy* and *delay*.

On the other hand, batching inputs can effectively improve throughput for DNN inference tasks, due to better utilization of parallel computing resources on GPUs, and less data copy between CPU and GPU memories [139]. Because batching leads to higher processing latency in the inference phase, the scheduler must consider *batch size selection* as well to satisfy the requirements on both latency and throughput.

Our work takes both issues into consideration: The scheduler dynamically selects a batch size and a DNN model to process on a GPU in a cluster. Its goal is to maximize the real-time system performance that is a utility function of inference accuracy, E2E delay, and query deadline. The servers may be heterogeneous in the aspect of GPU types, which leads to a more complex machine dependent scheduling problem. We describe two approaches to schedule DNN inference tasks on a cluster of GPU servers. We start with a heuristic scheduler: it selects the minimal batch size that satisfies the throughput requirement to keep E2E delays low. Although the scheduler is simple and efficient, its performance is sensitive to careful parameter tuning, and it tends to frequently use faster DNN models that provide lower accuracy.

To overcome the drawbacks, we further propose an advanced approach based on deep reinforcement learning (RL) [102]. Our RL scheduler uses real-time system status as the input to a policy DNN model that selects scheduling actions, which is trained to maximize QoS of query responses. To reduce the scheduling overhead, it performs actions on demand instead of running at a fixed time step. We conduct simulation-based experiments to evaluate the schedulers, and show that the RL approach achieves better performance.

7.2 Background

Adaptive batching: The adaptive batching problem for data analytics workloads running on processors has been studied in [44]. It presents an online algorithm that minimizes the latency while satisfying the throughput requirement, under varying data ingestion rates and other dynamic operation conditions. The algorithm learns the characteristics of workloads from the

statistics obtained by profiling the prior data, and uses numerical optimization techniques to find the optimal batch size. DjINN [63] improves DNN inference throughput on GPUs using batching and NVIDIA Multi-Process Service (MPS) [14] based techniques. However, adaptive batching is not adopted in their work. To dynamically adjust batch size of DNN inference workloads, Clipper [40] adopts two approaches: an additive-increase-multiplicative-decrease (AIMD) approach, and another approach that leverages estimated 99th-percentile latency as a function of batch size. The algorithm searches for the optimal batch size that maximizes throughput while meeting the latency constraint. The previous works consider the optimizations for accuracy, throughput and delay separately. In this work, we present new approaches that combine these metrics into a unified performance evaluation metric for adaptive batching.

Scheduling Video Analytics Queries: VideoStorm [160] presents scheduling techniques for video analytics queries that consider trade-offs between resource overhead, response quality, and processing delay. It uses an offline profiling phase to model the resource-quality profile of queries. The scheduling objectives that combine quality and delay are expressed as utility scores. In the online scheduling phase, it periodically adjusts resource allocation, query placement, and configurations, to either maximize the minimum utility or the sum of utilities, using heuristic approaches. In addition to scheduling queries in a single cluster, VideoEdge [72] further takes hierarchical clusters with diverse computing capabilities and network bandwidths into consideration. The framework dynamically places processing modules of a query across clusters, and selects one from the implementation alternatives and configuration knobs for the query to maximize the average accuracy. These works are evaluated using vision workloads on CPUs. Optimization techniques such as adaptive batching for DNN inference tasks on GPUs are not studied.

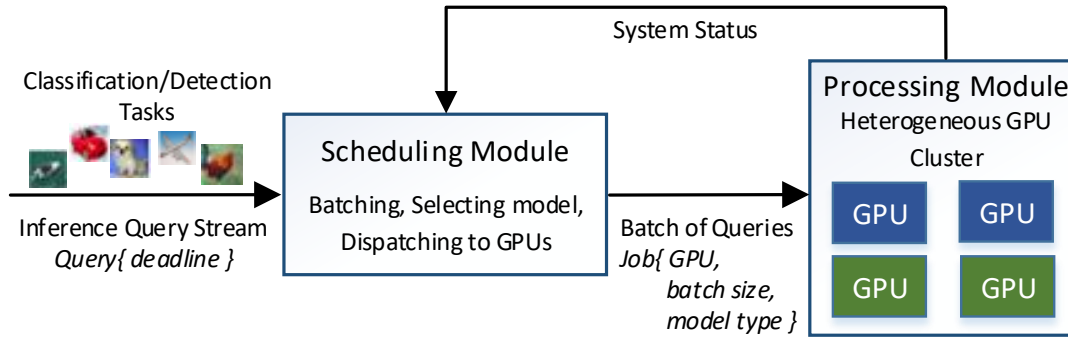


Figure 7.1. The architecture of DNN serving system. The system receives a stream of incoming queries that run DNN inference tasks. The scheduling module buffers the queries, and submits batches of queries to be processed by the processing module. It selects the batch sizes and the model type dynamically to optimize the response performance. The processing module is a cluster of GPUs that can be heterogeneous.

7.3 DNN Inference Server

7.3.1 System Overview

The system model is shown in Figure 7.1 where a server processes a continuous stream of incoming DNN inference queries sent by clients. A query comes with soft and hard deadlines (t_{ddl}^s and t_{ddl}^h) specified by the application, which give the time budget to process the query (described in Section 7.3.3). We assume that the client clock is synchronized to the server, as clock synchronization is generally available nowadays, even on mobile devices. The *processing module* in Figure 7.1 is a GPU cluster that may contain different types of GPUs. Each GPU processes batches of queries sequentially and returns the inference results. The *scheduling module*, which is the core of this work, packs queries as a batch and selects the suitable DNN model to run on GPUs.

7.3.2 The Impact of Batch Size

First we illustrate the impact of batching on processing throughput of DNNs with measurement data. We denote the forward propagation delay of a DNN as $D_{B,M,G}$, where B is the batch size, M is the DNN model, G is the type of GPU. There are generally two ways to speed

up a model: training a simpler model with fewer parameters, or simplifying the original model using approximation techniques such like matrix factorization, pruning, and quantization [60]. This work considers the former approach and uses YOLO [131], a model for real-time object detection, as the testcase. Two models are used: the *full* model achieves a mean average precision (mAP) of 63.4% on the PASCAL VOC dataset [48] and the mAP is 57.1% for the less complex *tiny* model [23]. We measure $D_{B,M,G}$ on AWS g2.2xlarge (g2) and more powerful p2.xlarge (p2) GPU instances.¹ The measured throughputs in queries per second (QPS) are given in Table 7.1, with B ranging from 1 to 64, $M \in \{\text{tiny}, \text{full}\}$, and $G \in \{\text{g2}, \text{p2}\}$. It shows that batching effectively improves the throughputs.

Table 7.1. DNN inference throughputs of tiny and full YOLO models (QPS). It shows that throughput can be evidently improved by using a larger batch size.

G	M	B=1	2	4	8	16	32	64
g2	tiny	25.6	35.7	51.9	66.1	73.7	84.7	91.2
	full	9.4	9.4	12.3	14.7	16.0	16.8	-
p2	tiny	23.8	35.7	63.5	90.9	109.6	131.1	141.9
	full	14.5	18.9	24.2	28.4	31.5	33.9	34.6

7.3.3 Performance Metrics of Query Response

Utility functions are widely used to express the benefit from performance gains in computer systems [160]. We use a time-utility function (TUF) V to represent the influence of delay on response quality. The quality of a query response (Q), defined as a function of the end time of query processing (t_{end}), the deadlines ($t_{ddl}^{s,h}$) and the DNN model (M), is computed as:

$$Q(t_{end}, t_{ddl}^{s,h}, M) = \kappa_M \cdot V(t_{end}, t_{ddl}^{s,h}), \quad (7.1)$$

where $\kappa_M \in (0, 1]$ is a factor to penalize less accurate models. The scheduler is designed to maximize the real-time response quality Q , therefore it is essential to select κ_M and V that accurately model the application performance. Choosing a proper TUF for time-sensitive

¹GPU type is NVIDIA GRID for g2.2xlarge and NVIDIA GK210 for p2.xlarge.

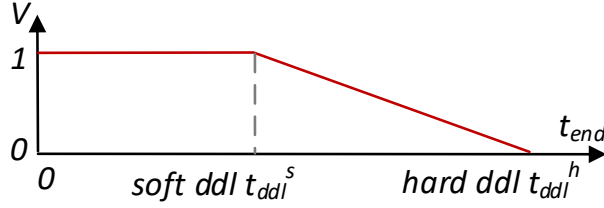


Figure 7.2. Utility of query response as a function of time delay $V(t_{end}, t_{ddl}^{s,h})$. The utility value is 1 before a soft deadline t_{ddl}^s , and then decreases linearly to 0 at a hard deadline t_{ddl}^h .

applications has been extensively studied by the real-time system community [128].

The TUF (V) used in this work is illustrated in Figure 7.2: the utility value of a response is 1 if t_{end} meets the soft deadline, otherwise it decreases linearly until zero at the hard deadline. This work does not consider downstream latency that is unknown at the scheduling time. Timecard [129] details a solution to predict downstream latency based on response content and network condition.

7.4 Scheduler Design

Scheduling algorithms based on utility scores typically maximize the minimum utility or sum of utilities [160]. In this work, our schedulers optimize sum of utility scores of queries in a continuous stream of batches or in continuous time windows, to consider a stream processing system. We design two QoS-aware schedulers: a simple heuristic scheduler and a more advanced scheduler using deep reinforcement learning. Both approaches prioritize the queries with closer deadlines to avoid missing deadlines. For a heavily loaded server with a high query arrival rate, it is inefficient to order every query based on deadline in the query buffer. As an alternative, we approximate query ordering by quantizing time into discrete bins, as shown in Figure 7.3. The duration of each bin is Δt . A query with soft deadline t_{ddl}^s is allocated to bin i with $i = (t_{ddl}^s - t_{init}) / \Delta t$, where t_{init} is the system initialization time. The bins are created and deleted dynamically. The scheduler fetches queries starting from the bins with the smallest indexes.

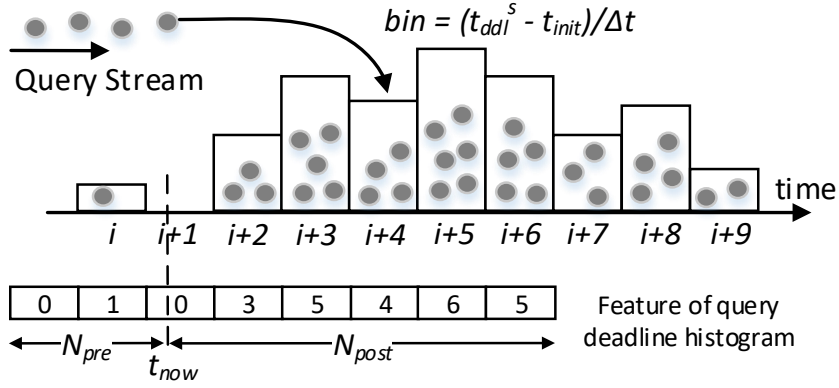


Figure 7.3. Buffering incoming queries in bins to approximately order queries according to deadline t_{ddl}^s . The number of queries in bins represents the urgency of all buffered queries, which is used as a feature in our reinforcement learning based scheduler (Section 7.4.2).

7.4.1 Simple Scheduling by Heuristic

The heuristic scheduler always selects the smallest batch size (B) that satisfies the requirement on throughput:

$$B \geq [\alpha \cdot (D_{B,M,G} \cdot R_t) + (1 - \alpha) \cdot N_t] \cdot \beta \cdot \xi_G, \quad (7.2)$$

where R_t is the query arrival rate at time t , and $\xi_G \in (0, 1]$ is the portion of queries allocated to GPU G . The term $D_{B,M,G} \cdot R_t$ represents the number of incoming queries. N_t is the total number of buffered queries. $\alpha \in [0, 1]$ and $\beta > 0$ are tuning parameters. The scheduler allocates queries to GPUs proportionally to processing capability. In this work we use the throughput with $B = 8$ as the processing capability of a GPU, and obtain ξ_G accordingly. The scheduler uses the most accurate DNN model in default. If the batch size is not less than a threshold $B_{thrd} = 8$, in order to improve QoS by reducing delay, the scheduler checks all simplified model types and selects the model maximizing the total quality value of this batch $Q_B = \kappa_M \cdot \sum_{q \in B} Q(t_{now} + D_{B,M,G}, t_{ddl}^{s,h}, M)$.

7.4.2 Advanced Scheduling by Deep Reinforcement Learning

In reinforcement learning, at time t , the agent observes the state s_t , chooses the action a_t , and obtains a reward r_t with this action. The learning process aims at maximizing the expected cumulative reward: $\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t]$, where $\gamma \in (0, 1]$ is the discount factor of future reward. The agent selects actions based on a policy $\pi(s, a)$ that gives the probability to perform a in state s . A deep RL agent uses a DNN to approximate the policy $\pi(s, a)$ (DNN details are in Section 7.5.2). To customize this deep RL agent in our scheduling problem, here we describe state, action, and reward. We follow the routine of a policy gradient RL algorithm to train the model [140].

State represents the run-time status of the system, including the following four parts:

Histogram of query deadlines: We use a histogram of query deadlines as the feature representing urgency of buffered queries. The feature vector is obtained from the sizes of query bins. As shown in Figure 7.3, at the current time t that corresponds to the bin i , the histogram feature comprises of N_{pre} bins before and N_{post} bins after the bin i . The size is zero for a non-existent bin, hence this feature has a fixed length $N_{pre} + N_{post}$. Because the histogram state has been truncated to a fixed length, it may be less than the real total number of buffered queries. Thus, we add the total number of buffered queries as an additional feature.

Arriving rate: The state contains the query arrival rate (R_t) to estimate the number of incoming queries in future.

GPU type: The scheduler must consider the processing capability of the GPU that processes the batch. We use a binary vector to represent the GPU type: an element e of the vector is 1 if the GPU matches the type that e is associated with, otherwise e is 0.

Availability of GPUs: When dispatching a batch to a GPU, the optimal batch size is influenced by the availability of other GPUs. We represent availability as the expected rest computing time of each device ($t_{end} - t_{now}$), where t_{end} is estimated using the measured mean computing time (see Table 7.1). The state includes the availabilities of all GPUs.

An **action** selects a batch size B and a model type M , then runs the selected model with

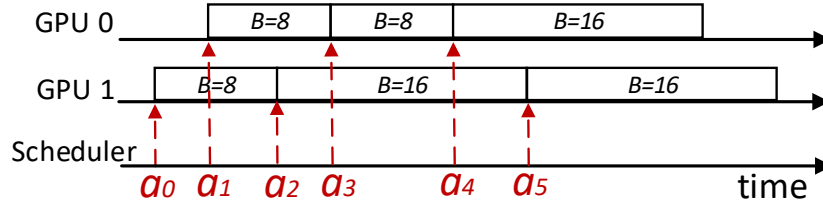


Figure 7.4. The reinforcement learning scheduler takes scheduling actions at dynamic time points when a GPU becomes available instead of at fine discrete points to reduce the scheduling overhead.

the batch of queries on a GPU. For a heterogeneous cluster, it is possible that an action is feasible for some types of GPUs but not for the others, for example, out of memory may happen when it runs a complex model with a large batch on a GPU with low memory capability. The scheduler automatically selects a smaller batch size when the action is not feasible.

The **reward** is defined as the average quality (Equation 7.1) of the latest W query responses. It represents the system’s real-time QoS. Consider that several GPUs may process queries in parallel, for the reward of the action a_t , the window W excludes any batches generated after t , to eliminate the noise on response quality caused by other GPUs.

Reinforcement learning typically computes actions at regular time intervals (*e.g.*, [102]). The interval should be short enough for the scheduler to perform timely actions. For example, in this scheduling problem, the inference delay of the smallest batch $B = 1$ is around 60ms, the scheduling interval should be even smaller to dispatch queries to newly available GPUs quickly. Using a small fixed scheduling interval exerts high computation overhead. To resolve this problem, as shown in Figure 7.4, we compute the next action to take using the policy DNN model only when there are free GPUs. It leads to a large reduction of the scheduling overhead when incoming workloads keep all GPUs busy. In the evaluation using a single GPU (described in Section 7.5.2), taking actions at dynamic time points needs only 3% action computations compared with using a fixed interval of 10ms.

7.5 Evaluation

7.5.1 Experiment Setting

We emulate the DNN inference server (Figure 7.1) on a simulation platform. The simulator generates newly arriving queries, schedules new batches of queries to run, and checks whether each GPU has finished the previous batch. A query is a YOLO object detection task that uses either the full or the tiny model. We use $\kappa_{full} = 1$ and $\kappa_{tiny} = 0.5$ for the models. The processing delay of a batch ($D_{B,M,G}$) is generated by a normal distribution fitting the measurement data (Table 7.1). The query stream is generated from a web query trace provided by the SogouQ query log dataset [22]. The QPS of query stream over a day is plotted in Figure 7.5. We generate new queries every 10ms. Because the timestamp resolution of the original query log is 1s, queries are allocated to finer 10ms intervals using a normal distribution. For each query, there is an upstream network delay before it arriving at the server. We consider a mobile computing scenario where clients send queries to the server via LTE networks. The delays are generated from a *Pareto* distribution which is fitted to LTE delay data produced by the Mahimahi tool [111]. Deadlines $t_{ddl}^s = 1s$ and $t_{ddl}^h = 2s$ are applied to all queries.

A baseline scheduler (denoted as *fixed*) is tested for performance comparison. It always batches all buffered queries to run on a GPU, with 32 as the maximal batch size. We use the query data in the time range from 9hr to 12hr (Figure 7.5) as the training data. The continuous query log is split into training jobs with a duration of 10min to train the RL scheduler. The training data is also used to optimize the parameters of the heuristic scheduler. Because the scheduling problem is trivial at low load level (time 0hr to 9hr), we are mostly interested in the scheduling performance with high load. Hence, the evaluation is done using 4 testing jobs generated from the time 12hr to 24hr, each lasts for 3hr (*e.g.*, see Table 7.2).

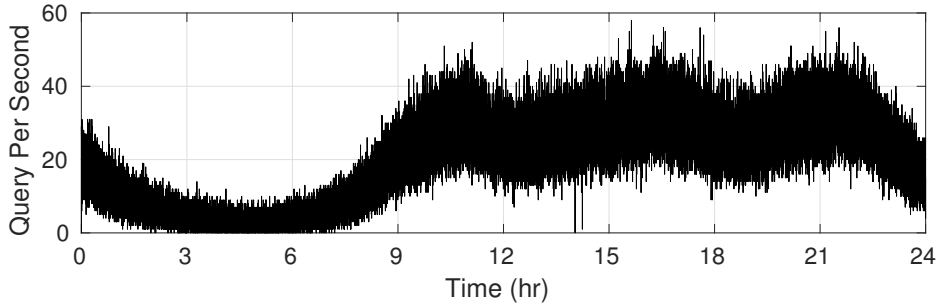


Figure 7.5. QPS of web queries in SogouQ dataset. We use the QPS values to generate queries to our DNN serving system in our emulations.

7.5.2 Performance of Schedulers

Single GPU: We first evaluate the schedulers using a processing module with one GPU (p2). Both schedulers use batch sizes as given in Table 7.1, and the RL scheduler has $B = 0$ as an additional option. The bin size of query buffer is $\Delta t = 200\text{ms}$. The arrival rate of queries (R_t) is measured in a sliding window of 5s. The heuristic schedule use the parameters $\alpha = 0.5$ and $\kappa = 0.5$ that give the highest mean response quality on the training data. The policy model of the RL scheduler has 1 fully connected hidden layer with 64 neurons. The histogram of deadlines feature in the state uses $N_{pre} = 4$ and $N_{post} = 6$ (Section 7.4.2). The reward is measured using the qualities of last $W = 1000$ queries.

The scheduling performance is measured with two metrics: (1) mean response quality (Equation 7.1) over 3 hours, given in Table 7.2 with standard deviation (SD); (2) P_{tiny} , the percentage of queries that use the tiny model and have lower inference accuracy, given in Table 7.3. The fixed scheduler that does not adapt batch size has the lowest response quality. The heuristic scheduler effectively improves the mean quality, and the RL scheduler achieves better performance. The Cumulative Distribution Functions (CDFs) of end time relative to soft deadline ($t_{end} - t_{ddl}^s$) of all queries are plotted in Figure 7.6a. It illustrates that compared with the heuristic approach, the RL approach has larger response delays, whereas it achieves higher quality by using the tiny model for less times (as in Figure 7.6b). The scheduling actions performed by the heuristic and RL schedulers in a duration of 10min are visualized in Figure 7.7.

Table 7.2. Mean (SD) Response Quality of Single GPU.

Scheduler	12-15hr	15-18hr	18-21hr	21-24hr
Fixed	0.883 (0.204)	0.818 (0.235)	0.860 (0.218)	0.863 (0.218)
Heuristic	0.968 (0.117)	0.915 (0.185)	0.950 (0.146)	0.939 (0.161)
RL	0.978 (0.084)	0.931 (0.149)	0.961 (0.114)	0.949 (0.132)

Table 7.3. Portion of tiny models P_{tiny} (%) of Single GPU.

Scheduler	12-15hr	15-18hr	18-21hr	21-24hr
Fixed	20.2	33.8	25.0	25.4
Heuristic	5.5	16.3	9.2	11.7
RL	2.0	8.2	4.1	6.1

Heterogeneous GPU Cluster: To evaluate the schedulers for more servers, we use a cluster of 5 g2 GPUs and 5 p2 GPUs. The query arrival rate is scaled up by 8 times in the simulation to fill up the processing capability of the cluster. The policy DNN model of the RL agent has the same hidden layer as the model for one GPU. The results are given in Table 7.4 and 7.5. We find that similar to the single GPU case, the RL scheduler delivers higher response quality by using the tiny YOLO model less frequently.

Table 7.4. Mean (SD) Response Quality of Multi-GPUs.

Scheduler	12-15hr	15-18hr	18-21hr	21-24hr
Fixed	0.820 (0.231)	0.767 (0.242)	0.801 (0.237)	0.813 (0.234)
Heuristic	0.948 (0.148)	0.883 (0.210)	0.925 (0.176)	0.916 (0.186)
RL	0.955 (0.139)	0.895 (0.202)	0.933 (0.168)	0.922 (0.179)

7.6 Summary

This chapter examines the QoS-aware scheduling problem for DNN inference workloads. We describe a simple heuristic scheduler and a more advanced scheduler based on deep reinforcement learning. In addition, we propose to perform actions on demand at dynamic points in time, which greatly reduces the scheduling overhead. Through simulation experiments, we demonstrate that the RL scheduler achieves higher mean response quality and uses the simplified DNN model less frequently.

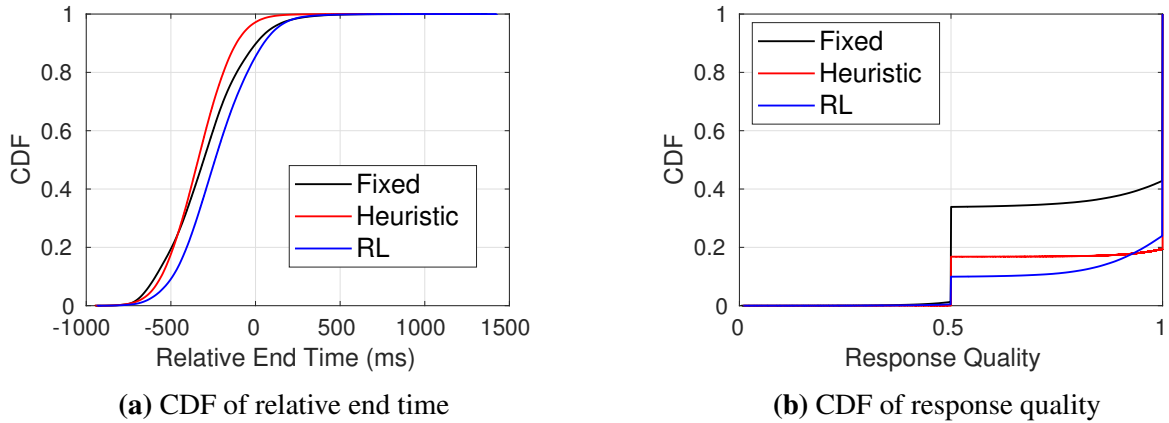


Figure 7.6. Performance comparison of fixed, heuristic and RL schedulers. We observe that (a) the RL scheduler has larger response delays, and (b) it attains higher response qualities by using the tiny model for less times.

Table 7.5. Portion of tiny models P_{tiny} (%) of Multi-GPUs.

Scheduler	12-15hr	15-18hr	18-21hr	21-24hr
Fixed	32.6	43.5	36.5	34.4
Heuristic	9.6	22.8	14.4	16.4
RL	8.2	20.6	12.8	14.8

This chapter, in full, is a reprint of the material as it appears in Zhou Fang, Tong Yu, Ole J. Mengshoel, and Rajesh K. Gupta. “QoS-Aware Scheduling of Heterogeneous Servers for Inference in Deep Neural Networks”, in Proceedings of the 2017 ACM Conference on Information and Knowledge Management (CIKM ’17), 2017. The dissertation author was the primary investigator and author of this paper.

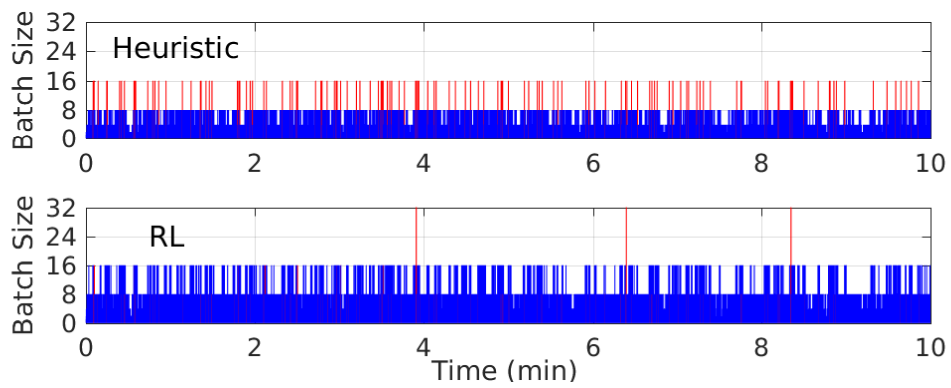


Figure 7.7. The timeline plots of scheduling actions in adaptive batching. Actions using full model are plotted in blue and tiny model in red. Compared to the heuristic scheduler, the RL scheduler uses the tiny model less frequently and thus attains higher response qualities (Table 7.2).

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This dissertation explores multi-tenant mobile offloading frameworks for real-time computer vision workloads, in particular DNN inference tasks. By presenting a complete system implementation covering APIs, scheduling algorithms, and distributed systems, we demonstrated that multiple clients running various applications can be served concurrently in a cloudlet with high resource utilization. Moreover, we have discovered several bottlenecks in existing offloading frameworks and presented our solutions described below:

- We designed mobile offloading APIs to manage mobile data on the server side in the form of in-memory key-value pairs, and program applications to offload as DAGs of queries that process mobile data (Chapter 3). High-level APIs using a data flow model that partition an application across mobile devices, cloudlet, and cloud servers are implemented as well (Chapter 6). These APIs can simplify application development and get rid of redundant data transfers as well as duplicated queries.
- We implemented serving systems for DNNs with different architectures for diverse vision tasks, *e.g.*, object detection, tracking, scene graph detection, and video description (Chapter 3). To improve GPU utilization in a cloudlet cluster, we designed schedulers that co-locate RT and non-RT tasks on shared GPUs, using a new predictive and planning-ahead

based approach to avoid resource contentions. It attains a high GPU utilization level while providing low delays for RT tasks (Chapter 4).

- For workloads processed on CPUs, we presented a two-phase Plan-Schedule approach to mitigate resource contentions that increase the E2E delays (Chapter 5). The planning phase predicts future tasks from all clients to estimate contentions. It then devises start times of future tasks to remove or reduce contentions. The scheduling phase dispatches incoming tasks to the best server that minimizes contentions. The planning and scheduling methods are enabled by new capabilities of task execution time prediction, network delay estimation, and clock synchronization.
- We designed scheduling algorithms to dynamically the adjust batch size and select model variant for DNN inference tasks (Chapter 7). Our schedulers use a utility function of response delay and inference accuracy as the QoS metric to optimize. A simple and effective heuristic approach that attains low response delay while satisfying the throughput requirement is presented. Then we described an advanced deep reinforcement learning approach that learns to schedule from experience and achieves higher QoS values.

Our work provides solutions to serve real-time vision applications on multi-tenant cloudlet servers with a high resource utilization. The systems that we implemented could facilitate the ubiquitous deployment of emerging vision and cognition applications on server-backed mobile platforms in future.

8.2 Limitations and Future Work

Although we have tackled several essential problems in offloading emerging mobile vision applications, because of the complexities that lie in every layer throughout the system stack, there are several limitations in our current work.

Our approaches to mitigate multi-tenant resource contention depend upon the capability to predict the timing of future workloads on servers (Chapter 4 and 5), which is enabled by

network delay estimation and clock synchronization. Therefore, our systems perform the best on local servers that are connected to mobile clients via high-bandwidth, low-latency WiFi networks. It limits our systems, *i.e.*, DeepQuery and ATOMS, to be used for remote servers in cloud and cellular networks. Another limitation is that we do not consider the scenario that the network bandwidth is not enough for serving a group of clients, which should be addressed in the future work. In addition, our DNN serving systems, as well as scheduling algorithms, are designed for GPUs and can not support other hardware accelerators such as FPGAs [51] and ASICs [78].

Here we describe a few specific research questions to be answered in our future work and potential approaches to improve our systems:

DNN Model Placement: We manually place DNN models on servers using a configuration file in the current DeepQuery implementation. However, automating model placement is essential to fully utilize server resources. To place DNN models onto GPUs, we should consider the following constraints: (1) The memory overheads of all models served on a GPU must be within the limits of device memory; (2) The number of GPUs that run a DNN model depends on the rate of the queries using that model; (3) The decision to co-locate different DNN models on shared GPUs should consider the timing requirements of the applications that use the models. For example, co-locating models for RT and non-RT queries helps improve resource utilization. However, the co-location method may not work if RT queries are allocated to shared workers, because they are both urgent and are competing resources. Online model placement techniques are to be investigated in the future work.

Use of DeepQuery in the Cloud: Although the DeepQuery framework is designed for local servers, it is useful for deploying DNNs in the cloud in several aspects. First, the APIs based on data operations and DNN queries on data are useful for offloading complex vision applications to the cloud resources as well. Second, the optimization techniques that we studied for various DNN models are general, and thus should apply to other serving systems. Compared to the cloudlet setting (*i.e.*, local servers) considered in this work, the scale of cloud computing platforms such as AWS is, of course, much larger, with possibly high query rates from clients.

DeepQuery can be used inside individual compute blocks in larger clusters with load balancers and schedulers that are implemented in upper layers.

The main limitation, however, is that the scheduling methods related to predictions of future query arriving times may not work efficiently under large network uncertainties to the cloud, because of traffics through the public networks. To maintain low levels of contentions when co-locating non-RT and RT queries, we need to use the upper bounds of network delays in the prediction methods. This would lower the GPU utilization because of the over-estimated network delays. The use of DeepQuery in the cloud needs to be verified with more evaluations in the future work.

Combining DeepQuery with Conventional Serving Systems: Going beyond cloudlets, DeepQuery can also be backed by cloud DNN serving systems [3, 4, 6]. In this *mobile-edge-cloud* deployment, DeepQuery processes RT queries on local servers, and a portion of delay-tolerant queries as well, in order to fully utilize local compute resources. It can process delay-tolerant queries that have tight data dependency on RT queries, thus reducing the amount of data transferred to the cloud, while the remaining queries can be sent to the cloud for processing.

CPU Workload Scheduling in ATOMS: Supporting GPUs in the ATOMS scheduler to reduce resource contention lies in the future work. Moreover, for a cluster with heterogeneous computing resources, *e.g.*, different processor types and numbers of cores, the future work of ATOMS is to improve the E2E delay for tasks with long upstream network latency by scheduling them on more powerful workers. The tail service delay can thus be improved.

Admission control is necessary for guaranteeing server responsiveness. Because ATOMS is able to maintain low E2E delays when the total CPU utilization is as high as over 80%, keeping a low processor utilization is not sufficient for admission control in ATOMS. The measured percentiles of offloading interval and the cost value of violating interval SLAs (Section 5.6.5) can be used as additional metrics, because dragged intervals can indicate that total reservations exceed the cluster's processing capability.

Appendix A

Programming Customized Microservices in DeepQuery

DeepQuery provides Python APIs to deploy customized computer vision algorithms or DNN models as microservices. We provide the details of DeepQuery APIs (Chapter 3) to program customized microservices using object detection as example.

A.1 Microservices

A microservice must provide a `run(data_list, method)` method to process input data to implement the abstract class `Microservice`. In the argument list, `data_list` is a list of input data and `method` is an identifier of the method to invoke provided by the microservice. For example, the Python class that implements an object detection microservice includes an initialization method to load a DNN model (`dnn_model`) from a configuration file (`configs`), and a `run` method to invoke DNN inference by calling `dnn_model.detect`.

Listing A.1. Implementing `run` method for object detection microservice `OBJECT_DET`.

```
1 # A microservice for object detection.
2 class ObjectDetection(Microservice):
3
4     def __init__(self, configs):
5         self.dnn_model = load_dnn_model(configs)
6
```

```

7 def run(self, data_list, method):
8     self.dnn_model.detect(*data_list)

```

A microservice can provide multiple methods to invoke which are selected using the argument method. For example, an FRCNN object detection model can be divided into three stages: RPN, CLS, and POST (Section 3.3.5). A microservice for this model can implement the following run method to process one of the three stages.

Listing A.2. Implementing run method for object detection microservice using data parallelization OBJECT_DET_PARALLEL.

```

1 # A microservice for object detection using data parallelization.
2 class ObjectDetectionParallel(Microservice):
3
4     def run(self, data_list, method):
5         if method == M_OBJECT_DET_RPN:
6             return self.dnn_model.run_rpn(*data_list)
7         elif method == M_OBJECT_DET_CLS:
8             return self.dnn_model.run_cls(*data_list)
9         elif method == M_OBJECT_DET_POST:
10            return self.dnn_model.run_postprocess(*data_list)
11        else:
12            logging.fatal("Method not supported.")

```

As described in Section 3.5.2, a query contains the identifiers of microservice and method to be processed correctly.

A.2 Creating Jobs for Queries

DeepQuery creates a job for each runnable query, which is executed by the JobRunner module (Section 3.6.1). A microservice must come with a method for creating jobs from queries. Listing A.3 gives an example of object detection queries. It creates a BasicJob that contain a single stage to be processed by the aforementioned microservice OBJECT_DET. The parameter

input_stages of a stage is the list of predecessor stages to obtain input data. The stage name “input” means that the input data are from the query input (data_list).

Listing A.3. Creating job for object detection query.

```
1 def _create_obj_detect_job():
2     return BasicJob(stage=Stage(name="det", microservice=OBJECT_DET,
3         input_stages=["input"]) )
```

To use data parallelization, a DAGJob containing RPN, CLS, and POST stages is created instead, as given in Listing A.4. The stages are processed by invoking corresponding methods of the microservice OBJECT_DET_PARALLEL. The output of DAGJob is a flattened list containing outputs of the stages given in the parameter output_stages.

For a normal stage Stage, the parameter transform_fn provides a method to transform the outputs of predecessor stages given by input_stages into the microservice input data (data_list). For a parallel stage ParallelStage running in parallel on multiple GPUs, data_transformer is a component that divide input data for the parallel running instances, and aggregates their outputs in a *fork-join* manner to obtain the final result.

Listing A.4. Creating job for object detection queries using data parallelization.

```
1 def _create_obj_detect_parallel_job():
2     def transform_fn(ims, rpn_output_dict, cls_output_dict):
3         return [ims, dict(rpn_output_dict.items() +
4             cls_output_dict.items())]
5
6     return DAGJob(output_stages=["det"],
7         stages=[Stage(name="rpn", microservice=OBJECT_DET_PARALLEL,
8             method=M_OBJECT_DET_RPN,
9             input_stages=["input"]),
10        ParallelStage(name="cls", microservice=OBJECT_DET_PARALLEL,
11            method=M_OBJECT_DET_CLS,
12            input_stages=["rpn"],
```

```
13     data_transformer=DataTransformer()),
14 Stage(name="det", microservice=OBJECT_DET_PARALLEL,
15     method=M_OBJECT_DET_POST,
16     input_stages=["input", "rpn", "cls"],
17     transform_fn=transform_fn)] )
```

Bibliography

- [1] *AWS Auto Scaling Groups*. [Online; accessed 10-Oct-2018].
- [2] *AWS Instance Pricing*. [Online; accessed 10-Oct-2018].
- [3] *AWS Machine Learning*. <https://aws.amazon.com/machine-learning/>. [Online; accessed 16-July-2018].
- [4] *Azure Machine Learning*. <https://azure.microsoft.com/overview/machine-learning/>. [Online; accessed 16-July-2018].
- [5] *Docker*. <https://www.docker.com/>. [Online; accessed 16-July-2018].
- [6] *Google Cloud Machine Learning*. <https://cloud.google.com/products/machine-learning/>. [Online; accessed 16-July-2018].
- [7] *Google Cloud Vision*. <https://cloud.google.com/vision>. [Online; accessed 16-July-2018].
- [8] *gRPC*. <https://grpc.io/>. [Online; accessed 16-July-2018].
- [9] *Hadoop*. <http://hadoop.apache.org/>. [Online; accessed 16-July-2018].
- [10] *Kubernetes*. [Online; accessed 16-July-2018].
- [11] *Kubernetes Horizontal Pod Autoscaler*. [Online; accessed 10-Oct-2018].
- [12] *Latest Innovations in TensorFlow Serving*. <https://ai.googleblog.com/2017/11/latest-innovations-in-tensorflow-serving.html>. [Online; accessed 16-July-2018].
- [13] *Microsoft Cognitive Services*. <https://www.microsoft.com/cognitive-services>. [Online; accessed 16-July-2018].
- [14] *NVIDIA Multi-Process Service*. https://docs.nvidia.com/deploy/pdf/CUDA_Multi-Process_Service_Overview.pdf. [Online; accessed 16-July-2018].
- [15] *OpenALPR: Automatic License Plate Recognition*. [Online; accessed 10-Oct-2018].
- [16] *Redis*. [Online; accessed 16-July-2018].

- [17] Tensorflow Detection Model Zoo. https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md. [Online; accessed 16-July-2018].
- [18] *The Arduino board*. [Online; accessed 16-July-2018].
- [19] The AWS DeepLens. <https://aws.amazon.com/deeplens/>. [Online; accessed 16-July-2018].
- [20] The Google Glasses. <https://x.company/glass/>. [Online; accessed 16-July-2018].
- [21] *The IOIO board*. [Online; accessed 16-July-2018].
- [22] *The SogouQ dataset*. [Online; accessed 16-July-2018].
- [23] YOLO Models. <https://pjreddie.com/darknet/yolo/>. [Online; accessed 16-July-2018].
- [24] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, 2016. USENIX Association.
- [25] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.
- [26] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, and S. Whittle. The dataflow model: A practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, Aug. 2015.
- [27] T. Amert, N. Otterness, M. Yang, J. H. Anderson, and F. D. Smith. Gpu scheduling on the nvidia tx2: Hidden details revealed. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 104–115, Dec 2017.
- [28] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 185–198, Lombard, IL, 2013. USENIX.
- [29] P. Anderson, X. He, C. Buehler, D. Teney, M. Johnson, S. Gould, and L. Zhang. Bottom-up and top-down attention for image captioning and visual question answering. In *CVPR*, 2018.
- [30] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool. Speeded-Up Robust Features (SURF). *Computer Vision and Image Understanding*, 110(3):346–359, June 2008.

- [31] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, C. Y. Koo, L. Lew, C. Mewald, A. N. Modi, N. Polyzotis, S. Ramesh, S. Roy, S. E. Whang, M. Wicke, J. Wilkiewicz, X. Zhang, and M. Zinkevich. Tfx: A tensorflow-based production-scale machine learning platform. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 1387–1395, New York, NY, USA, 2017. ACM.
- [32] L. Bertinetto, J. Valmadre, J. F. Henriques, A. Vedaldi, and P. H. S. Torr. Fully-convolutional siamese networks for object tracking. In G. Hua and H. Jégou, editors, *Computer Vision – ECCV 2016 Workshops*, pages 850–865, Cham, 2016. Springer International Publishing.
- [33] S. Bhattacharya and N. D. Lane. Sparsification and Separation of Deep Learning Layers for Constrained Resource Inference on Wearables. In *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems (SenSys)*, pages 176–189, 2016.
- [34] G. Bradski. The Opencv library. *Dr. Dobb's Journal of Software Tools*, 2000.
- [35] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and efficient parallel processing of massive data sets. *Proc. VLDB Endow.*, 1(2):1265–1276, Aug. 2008.
- [36] Q. Chen, H. Yang, J. Mars, and L. Tang. Baymax: Qos awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 681–696, New York, NY, USA, 2016. ACM.
- [37] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan. Glimpse: Continuous, Real-Time Object Recognition on Mobile Devices. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2015.
- [38] Z. Chen, W. Hu, J. Wang, S. Zhao, B. Amos, G. Wu, K. Ha, K. Elgazzar, P. Pillai, R. Klatzky, D. Siewiorek, and M. Satyanarayanan. An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, SEC '17, pages 14:1–14:14, New York, NY, USA, 2017. ACM.
- [39] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: Elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, 2011.
- [40] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, 2017.
- [41] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2010.

- [42] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893 vol. 1, June 2005.
- [43] M. Dantone, L. Bossard, T. Quack, and L. van Gool. Augmented Faces. In *2011 IEEE International Conference on Computer Vision Workshop (ICCVW)*, 2011.
- [44] T. Das, Y. Zhong, I. Stoica, and S. Shenker. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing, SOCC '14*, pages 16:1–16:13, New York, NY, USA, 2014. ACM.
- [45] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.
- [46] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, June 2009.
- [47] A. Elgammal, D. Harwood, and L. Davis. Non-parametric model for background subtraction. In D. Vernon, editor, *Computer Vision — ECCV 2000*, pages 751–767, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.
- [48] M. Everingham, L. Van Gool, C. K. Williams, J. Winn, and A. Zisserman. The Pascal visual object classes (VOC) challenge. *International journal of computer vision*, 88(2):303–338, 2010.
- [49] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An Updated Performance Comparison of Virtual Machines and Linux Containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015.
- [50] F. Fleuret, J. Berclaz, R. Lengagne, and P. Fua. Multicamera People Tracking with a Probabilistic Occupancy Map. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30(2):267–282, Feb 2008.
- [51] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger. A configurable cloud-scale dnn processor for real-time ai. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, June 2018.
- [52] M. Garcia-Valls, T. Cucinotta, and C. Lu. Challenges in real-time virtualization and predictable cloud computing.
- [53] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo. DSP.Ear: Leveraging co-processor support for continuous audio sensing on smartphones. In *Proceedings of the 12th ACM Conference on Embedded Network Sensor Systems (SenSys)*, 2014.

- [54] P. Georgiev, N. D. Lane, K. K. Rachuri, and C. Mascolo. LEO: Scheduling sensor inference algorithms across heterogeneous mobile processors and network resources. In *Proceedings of the 22Nd Annual International Conference on Mobile Computing and Networking*, MobiCom '16, pages 320–333, 2016.
- [55] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: Code offload by migrating execution transparently. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.
- [56] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards Wearable Cognitive Assistance. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2014.
- [57] K. Ha, P. Pillai, W. Richter, Y. Abe, and M. Satyanarayanan. Just-in-time provisioning for cyber foraging. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 153–166, 2013.
- [58] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 243–254. IEEE, 2016.
- [59] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.
- [60] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 123–136, 2016.
- [61] W. Han, P. Khorrami, T. L. Paine, P. Ramachandran, M. Ba'eizadeh, H. Shi, J. Li, S. Yan, and T. S. Huang. Seq-nms for video object detection. *CoRR*, abs/1602.08465, 2016.
- [62] A. Haque et al. Towards vision-based smart hospitals: A system for tracking and monitoring hand hygiene compliance. In *Proceedings of the Machine Learning for Healthcare Conference*, 2017.
- [63] J. Hauswald, Y. Kang, M. A. Laurenzano, Q. Chen, C. Li, T. Mudge, R. G. Dreslinski, J. Mars, and L. Tang. Djinn and tonic: Dnn as a service and its implications for future warehouse scale computers. In *ISCA*, pages 27–40, 2015.
- [64] J. Hauswald, M. A. Laurenzano, Y. Zhang, C. Li, A. Rovinski, A. Khurana, R. G. Dreslinski, T. Mudge, V. Petrucci, L. Tang, and J. Mars. Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 223–238, 2015.

- [65] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask r-cnn. In *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 2980–2988, Oct 2017.
- [66] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, June 2016.
- [67] F. C. Heilbron, V. Escorcia, B. Ghanem, and J. C. Niebles. Activitynet: A large-scale video benchmark for human activity understanding. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 961–970, June 2015.
- [68] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI’11*, pages 295–308, Berkeley, CA, USA, 2011. USENIX Association.
- [69] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, Nov. 1997.
- [70] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.
- [71] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, and K. Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3296–3297, July 2017.
- [72] C.-C. Hung, G. Ananthanarayanan, P. Bodík, L. Golubchik, M. Yu, V. Bahl, and M. Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *Proceedings of the Third ACM/IEEE Symposium on Edge Computing, SEC ’18*. ACM, 2018.
- [73] L. N. Huynh, Y. Lee, and R. K. Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys ’17*, pages 82–95, New York, NY, USA, 2017. ACM.
- [74] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size. *arXiv:1602.07360*, 2016.
- [75] B. Iannucci, P. Tague, O. J. Mengshoel, and J. Lohn. Crossmobile: A Cross-Layer Architecture for Next-Generation Wireless Systems. *CMU Technical Report CMU-SV-14-001*, 2014.
- [76] V. Jacobson. Congestion Avoidance and Control. In *Symposium Proceedings on Communications Architectures and Protocols (SIGCOMM)*, pages 314–329, 1988.

- [77] P. Jain, J. Manweiler, and R. Roy Choudhury. OverLay: Practical mobile augmented reality. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2015.
- [78] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-I. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snellham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA '17*, pages 1–12, New York, NY, USA, 2017. ACM.
- [79] Y. Ju, Y. Lee, J. Yu, C. Min, I. Shin, and J. Song. SymPhoney: A Coordinated Sensing Flow Execution Engine for Concurrent Mobile Sensing Applications. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems (SenSys)*, 2012.
- [80] Y. Ju, C. Min, Y. Lee, J. Yu, and J. Song. An Efficient Dataflow Execution Method for Mobile Context Monitoring Applications. In *2012 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 116–121, 2012.
- [81] T. Kämäräinen, M. Siekkinen, A. Ylä-Jääski, W. Zhang, and P. Hui. Dissecting the end-to-end latency of interactive mobile video applications. In *Proceedings of the 18th International Workshop on Mobile Computing Systems and Applications, HotMobile '17*, pages 61–66, New York, NY, USA, 2017. ACM.
- [82] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '17*, pages 615–629, New York, NY, USA, 2017. ACM.
- [83] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang. ThinkAir: Dynamic Resource Allocation and Parallel Execution in the Cloud for Mobile Code Offloading. In *2012 Proceedings IEEE INFOCOM*, pages 945–953, 2012.
- [84] R. Krishna, Y. Zhu, O. Groth, J. Johnson, K. Hata, J. Kravitz, S. Chen, Y. Kalantidis, L.-J. Li, D. A. Shamma, M. S. Bernstein, and L. Fei-Fei. Visual genome: Connecting language and vision using crowdsourced dense image annotations. *Int. J. Comput. Vision*, 123(1):32–73, May 2017.

- [85] R. Krishna, Y. Zhu, O. Groth, J. Johnson, K. Hata, J. Kravitz, S. Chen, Y. Kalantidis, L.-J. Li, D. A. Shamma, M. S. Bernstein, and L. Fei-Fei. Visual genome: Connecting language and vision using crowdsourced dense image annotations. *International Journal of Computer Vision*, 123(1):32–73, May 2017.
- [86] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.
- [87] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83—97, 1955.
- [88] J. R. Kwapisz, G. M. Weiss, and S. A. Moore. Activity Recognition Using Cell Phone Accelerometers. *SIGKDD Explor. Newsl.*, 12(2):74–82, 2011.
- [89] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.
- [90] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *Proceedings of the 15th International Conference on Information Processing in Sensor Networks (IPSN)*, 2016.
- [91] N. D. Lane and P. Georgiev. Can Deep Learning Revolutionize Mobile Sensing? In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications (HotMobile)*, pages 117–122, 2015.
- [92] J. Leverich and C. Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems, EuroSys '14*, pages 4:1–4:14, New York, NY, USA, 2014. ACM.
- [93] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft coco: Common objects in context. In D. Fleet, T. Pajdla, B. Schiele, and T. Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 740–755, Cham, 2014. Springer International Publishing.
- [94] C. L. Liu and J. W. Layland. Tutorial: Hard real-time systems. chapter Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment, pages 174–189. IEEE Computer Society Press, Los Alamitos, CA, USA, 1989.
- [95] H. Liu. A measurement study of server utilization in public clouds. In *2011 IEEE Ninth International Conference on Dependable, Autonomic and Secure Computing*, pages 435–442, Dec 2011.
- [96] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *Proceedings of the*

16th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '18, pages 389–400, New York, NY, USA, 2018. ACM.

- [97] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. Ssd: Single shot multibox detector. In B. Leibe, J. Matas, N. Sebe, and M. Welling, editors, *Computer Vision – ECCV 2016*, pages 21–37, Cham, 2016. Springer International Publishing.
- [98] D. G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, Nov. 2004.
- [99] H. Lu, J. Yang, Z. Liu, N. D. Lane, T. Choudhury, and A. T. Campbell. The Jigsaw Continuous Sensing Engine for Mobile Phone Applications. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 71–84, 2010.
- [100] Y. Lu, A. Chowdhery, and S. Kandula. Optasia: A relational platform for efficient large-scale video analytics. In *Proceedings of the Seventh ACM Symposium on Cloud Computing, SoCC '16*, pages 57–70, New York, NY, USA, 2016. ACM.
- [101] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *IJCAI*, pages 674–679, 1981.
- [102] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *HotNets*, 2016.
- [103] K. Marzullo and S. Owicki. Maintaining the Time in a Distributed System. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 295–305, 1983.
- [104] A. Mathur, N. D. Lane, S. Bhattacharya, A. Boran, C. Forlivesi, and F. Kawsar. Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '17*, pages 68–81, New York, NY, USA, 2017. ACM.
- [105] O. J. Mengshoel, B. Iannucci, and A. Ishihara. Mobile Computing: Challenges and Opportunities for Autonomy and Feedback. In *Presented as part of the 8th International Workshop on Feedback Computing*, 2013.
- [106] D. Mills, J. Martin, J. Burbank, and W. Kasch. Network Time Protocol Version 4: Protocol and Algorithms Specification. RFC 5905 (Proposed Standard), June 2010.
- [107] E. Miluzzo, N. D. Lane, K. Fodor, R. Peterson, H. Lu, M. Musolesi, S. B. Eisenman, X. Zheng, and A. T. Campbell. Sensing Meets Mobile Social Networks: The Design, Implementation and Evaluation of the Cenceme Application. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (SenSys)*, pages 337–350, 2008.

- [108] C. Min, S. Lee, C. Lee, Y. Lee, S. Kang, S. Choi, W. Kim, and J. Song. PADA: Power-aware Development Assistant for Mobile Sensing Applications. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, pages 946–957, 2016.
- [109] M.-M. Moazzami, D. E. Phillips, R. Tan, and G. Xing. ORBIT: A smartphone-based platform for data-intensive embedded sensing applications. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks (IPSN)*, 2015.
- [110] S. H. Mortazavi, M. Salehe, C. S. Gomes, C. Phillips, and E. de Lara. Cloudpath: A multi-tier cloud computing framework. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing, SEC '17*, pages 20:1–20:13, New York, NY, USA, 2017. ACM.
- [111] R. Netravali, A. Sivaraman, K. Winstein, S. Das, A. Goyal, and H. Balakrishnan. Mahimahi: A Lightweight Toolkit for Reproducible Web Measurement. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, 2014.
- [112] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden. Wishbone: Profile-based Partitioning for Sensornet Applications. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.
- [113] S. Nirjon, R. F. Dickerson, P. Asare, Q. Li, D. Hong, J. A. Stankovic, P. Hu, G. Shen, and X. Jiang. Auditeur: A mobile-cloud service platform for acoustic event detection on smartphones. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys '13*, pages 403–416, 2013.
- [114] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *SOSP*, 2013.
- [115] C. S. Pabla. Completely Fair Scheduler. *Linux Journal*, 2009(184), Aug. 2009.
- [116] R. Panda, A. Bhuiyan, V. Murino, and A. K. Roy-Chowdhury. Unsupervised adaptive re-identification in open world dynamic camera networks. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1377–1386, July 2017.
- [117] P. Pandey and D. Pompili. MobiDiC: Exploiting the Untapped Potential of Mobile Distributed Computing via Approximation. In *2016 IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 1–9, 2016.
- [118] J. J. K. Park, Y. Park, and S. Mahlke. Chimera: Collaborative preemption for multitasking on a shared gpu. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15*, pages 593–606, New York, NY, USA, 2015. ACM.
- [119] J. J. K. Park, Y. Park, and S. Mahlke. Dynamic resource management for efficient utilization of multitasking gpu. In *ASPLOS*, pages 527–540, 2017.

- [120] V. Paxson, M. Allman, J. Chu, and M. Sargent. Computing TCP’s Retransmission Timer. RFC 6298 (Proposed Standard), June 2011.
- [121] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: Enabling Interactive Perception Applications on Mobile Devices. In *MobiSys*, 2011.
- [122] K. K. Rachuri, C. Mascolo, M. Musolesi, and P. J. Rentfrow. Sociablesense: Exploring the trade-offs of adaptive sampling and computation offloading for social sensing. In *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking*, MobiCom ’11, pages 73–84, 2011.
- [123] K. K. Rachuri, M. Musolesi, C. Mascolo, P. J. Rentfrow, C. Longworth, and A. Aucinas. EmotionSense: A Mobile Phones Based Adaptive Platform for Experimental Social Psychology Research. In *Proceedings of the 12th ACM International Conference on Ubiquitous Computing (UbiComp)*, pages 281–290, 2010.
- [124] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time systems. In *SPIE/ACM Conference on Multimedia Computing and Networking*, 1998.
- [125] M. Ramon, S. Caharel, and B. Rossion. The speed of recognition of personally familiar faces. *Perception*, 40(4):437–449, 2011.
- [126] C. Raposo, M. Antunes, and J. P. Barreto. Piecewise-Planar Stereoscan: Structure and motion from plane primitives. In *Proceedings of the 13th European Conference on Computer Vision (ECCV)*. 2014.
- [127] J. Rasley, K. Karanasos, S. Kandula, R. Fonseca, M. Vojnovic, and S. Rao. Efficient queue management for cluster scheduling. In *EuroSys*, 2016.
- [128] B. Ravindran, E. D. Jensen, and P. Li. On recent advances in time/utility function real-time scheduling and resource management. In *Proceedings of the Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, ISORC ’05, pages 55–60, 2005.
- [129] L. Ravindranath, J. Padhye, R. Mahajan, and H. Balakrishnan. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, 2013.
- [130] L. Ravindranath, A. Thiagarajan, H. Balakrishnan, and S. Madden. Code in the air: Simplifying sensing and coordination tasks on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, HotMobile ’12, pages 4:1–4:6, 2012.
- [131] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You Only Look Once: Unified, Real-Time Object Detection. In *Proceedings of the 29th IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

- [132] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(6):1137–1149, June 2017.
- [133] M. Saini, S. P. Venkatagiri, W. T. Ooi, and M. C. Chan. The Jiku mobile video dataset. In *MMSys*, 2013.
- [134] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The case for vm-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct 2009.
- [135] H. Shen, A. Balasubramanian, A. LaMarca, and D. Wetherall. Enhancing mobile apps to use sensor hubs without programmer effort. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing, UbiComp '15*, pages 227–238, 2015.
- [136] C. Shi, K. Habak, P. Pandurangan, M. Ammar, M. Naik, and E. Zegura. COSMOS: Computation Offloading As a Service for Mobile Devices. In *Proceedings of the 15th ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc)*, 2014.
- [137] P. Simoens, Y. Xiao, P. Pillai, Z. Chen, K. Ha, and M. Satyanarayanan. Scalable crowdsourcing of video from mobile devices. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 139–152, 2013.
- [138] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In *International Conference on Learning Representations*, 2015.
- [139] M. Song, Y. Hu, H. Chen, and T. Li. Towards pervasive and user satisfactory cnn across gpu microarchitectures. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, Feb 2017.
- [140] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, 1999.
- [141] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Computer Vision and Pattern Recognition (CVPR)*, 2015.
- [142] I. Tanasic, I. Gelado, J. Cabezas, A. Ramirez, N. Navarro, and M. Valero. Enabling preemptive multiprogramming on gpus. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 193–204, June 2014.
- [143] S. P. Tarzia, P. A. Dinda, R. P. Dick, and G. Memik. Indoor Localization Without Infrastructure Using the Acoustic Background Spectrum. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 155–168, 2011.

- [144] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*, 2013.
- [145] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy. Storm@twitter. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 147–156, New York, NY, USA, 2014. ACM.
- [146] A. Tumanov, T. Zhu, J. W. Park, M. A. Kozuch, M. Harchol-Balter, and G. R. Ganger. Tetrished: Global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *EuroSys*, 2016.
- [147] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.
- [148] S. Venugopalan, M. Rohrbach, J. Donahue, R. Mooney, T. Darrell, and K. Saenko. Sequence to sequence – video to text. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 4534–4542, Dec 2015.
- [149] P. Viola and M. Jones. Rapid Object Detection using a Boosted Cascade of Simple Features. In *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, 2001.
- [150] H. Wang and C. Schmid. Action recognition with improved trajectories. In *2013 IEEE International Conference on Computer Vision*, pages 3551–3558, Dec 2013.
- [151] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo. Simultaneous multikernel gpu: Multi-tasking throughput processors via fine-grained sharing. In *HPCA*, pages 358–369, 2016.
- [152] B. Wu, F. Iandola, P. H. Jin, and K. Keutzer. Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, July 2017.
- [153] J. Wu, C. Leng, Y. Wang, Q. Hu, and J. Cheng. Quantized convolutional neural networks for mobile devices. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4820–4828, June 2016.
- [154] Y. Wu, Y. Sui, and G. Wang. Vision-based real-time aerial object localization and tracking for uav sensing system. *IEEE Access*, 5:23969–23978, 2017.
- [155] D. Xu, Y. Zhu, C. B. Choy, and L. Fei-Fei. Scene graph generation by iterative message passing. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3097–3106, July 2017.

- [156] J. Xu, T. Mei, T. Yao, and Y. Rui. Msr-vtt: A large video description dataset for bridging video and language. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5288–5296, June 2016.
- [157] T. Yao et al. MSR Asia MSM at ActivityNet challenge 2017. <http://home.ustc.edu.cn/~panywei/paper/Activitynet17.pdf>, 2017. [Online].
- [158] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li. Lavea: Latency-aware video analytics on edge computing platform. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing, SEC '17*, pages 15:1–15:13, New York, NY, USA, 2017. ACM.
- [159] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing, HotCloud'10*, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.
- [160] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live video analytics at scale with approximation and delay-tolerance. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 377–392, Boston, MA, 2017. USENIX Association.
- [161] I. Zhang, A. Szekeres, D. Van Aken, I. Ackerman, S. D. Gribble, A. Krishnamurthy, and H. M. Levy. Customizable and extensible deployment for mobile/cloud applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.
- [162] T. Zhang, A. Chowdhery, P. V. Bahl, K. Jamieson, and S. Banerjee. The design and implementation of a wireless video surveillance system. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking, MobiCom '15*, pages 426–438, New York, NY, USA, 2015. ACM.
- [163] Y. Zhao, S. Szpiro, J. Knighten, and S. Azenkot. CueSee: Exploring Visual Cues for People with Low Vision to Facilitate a Visual Search Task. In *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp)*, pages 73–84, 2016.
- [164] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.