Spring 2022

System Abstractions for Scalable Application Development at the Edge

Bo Hu
Yale University Graduate School of Arts and Sciences, bo.hu.yins@gmail.com

Abstract

System Abstractions for Scalable Application Development at the Edge

Bo Hu

2022

Recent years have witnessed an explosive growth of Internet of Things (IoT) devices, which collect or generate huge amounts of data. Given diverse device capabilities and application requirements, data processing takes place across a range of settings, from on-device to a nearby edge server/cloud and remote cloud. Consequently, edge-cloud coordination has been studied extensively from the perspectives of job placement, scheduling and joint optimization. Typical approaches focus on performance optimization for individual applications. This often requires domain knowledge of the applications, but also leads to application-specific solutions. Application development and deployment over diverse scenarios thus incur repetitive manual efforts.

There are two overarching challenges to provide system-level support for application development at the edge. First, there is inherent heterogeneity at the device hardware level. The execution settings may range from a small cluster as an edge cloud to on-device inference on embedded devices, differing in hardware capability and programming environments. Further, application performance requirements vary significantly, making it even more difficult to map different applications to already heterogeneous hardware. Second, there are trends towards incorporating edge and cloud and multi-modal data. Together, these add further dimensions to the design space and increase the complexity significantly.

In this thesis, we present a novel framework to simplify application development and deployment over a continuum of edge to cloud. Our framework connects different dimensions of design considerations, corresponding to the application abstraction, data abstraction and resource management abstraction respectively.

First, our framework masks hardware heterogeneity with abstract resource types through containerization, and abstracts the application processing pipelines into generic flow graphs. Further, our framework supports a notion of degradable computing for application scenarios

at the edge that are driven by multimodal sensory input. Next, as video analytics is the killer app of edge computing, we include a generic data management service between video query systems and a video store to organize video data at the edge. We propose a video data unit abstraction based on a notion of distance between objects in the video, quantifying the semantic similarity among video data. Last, considering concurrent application execution, our framework supports multi-application offloading with device-centric control, with a userspace scheduler service that wraps over the operating system scheduler.

System Abstractions for Scalable Application Development at the Edge

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Bo Hu

Dissertation Director: Yang Richard Yang, Wenjun Hu

May 2022

Contents

List of Figures

List of Tables

Acknowledgments

I spent the unforgettable seven years at Yale. Lots of people make my Yale life a fantastic experience. First, I would like to thank Prof. Wenjun Hu and Prof. Richard Yang, my thesis advisors, for guiding this work. They gave me a great opportunity to pursue a Ph.D. degree at Yale. Their continuous research support enabled me to work on multiple exciting projects. I want to say special thanks to Prof. Richard Yang for his generous help during my dissertation submission process. Second, I want to thank my committee members, Prof. Lin Zhong, Prof. Anurag Khandelwal and Dr. Victor Bahl for their constructive feedback regarding my thesis. Third, I gratefully acknowledge the funding that supported the research which appears in this dissertation. In particular, my works are funded by the National Science Foundation under Grant No. CNS-1815115. Finally, I would like to sincerely thank my family, girlfriend, and friends. I cannot imagine going through this rewarding but sometimes stressful process without their unconditional love and support.

To my family

Chapter 1

Introduction

1.1 Applications at the edge

Recent years have witnessed an increasing number of Internet of Things (IoT) devices, ranging from smart thermometers to industrial robots. According to the Cellular IoT and LPWA Connectivity Market Tracker report [1], by the end of 2025, there will be more than 41.2 billion devices that have active network connections. 31 billion of them will be IoT devices. By analyzing the data generated by these devices, one can make better decisions. For example, real-time video analytics (e.g., collision detection [2] and traffic monitoring [3]), smart home (e.g., intruder alert [4] and smart lighting [5]), smart health (e.g., fall detection [6] and sleep monitoring [7]) and industrial automation (e.g., equipment maintenance [8] and manufacturing quality control [9]) are already seen in everyday life.

Many IoT applications are powered by edge computing, an emerging paradigm that extends computation, communication and storage capacities toward the network edge, either directly on IoT devices or on nearby computation servers [10]. Compared to local-only computation, edge computing overcomes the restrictions of limited computation capacity on IoT devices. Compared to moving workloads to remote servers, edge computing brings several benefits, including reducing network latency, reducing the backbone network overhead and preserving data privacy [11]. Edge devices range from end devices like smartphones and IP cameras to a cluster of servers located at the network edge. In other words, edge devices can include all but servers in the cloud.

Various support is needed across the software stack to build an application at the edge. For a general software stack, there are four layers, from top to bottom: application, middleware, OS and hardware. The middleware layer manages interactions between the application and the OS layers, providing more intuitive system abstractions to the application layer. A database is a simple example of such a middleware.

Applications at the edge typically have software stacks like those shown in Figure 1.1. There is no explicit middleware layer in the current edge computing landscape. Existing works have mainly focused on improving application performance on the edge via application-specific solutions [12–16], ignoring the programmability challenges that application developers might face. In other words, application developers still need a significant amount of domain-specific knowledge outside of that specific to their application to develop an application at the edge. For example, developers have to determine the right amount of computation resources they want before actually deploy their applications on the edge. Application development at the edge is still unscalable.

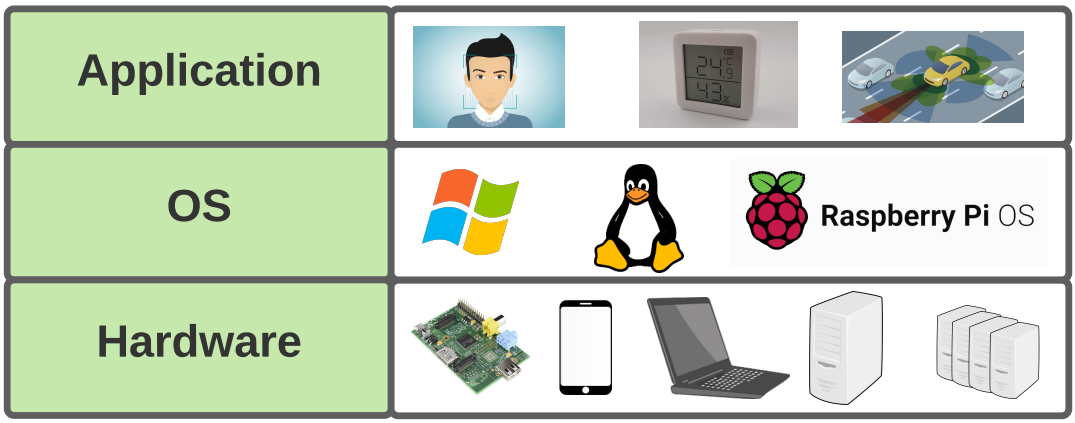


Figure 1.1: A typical software stack of applications running at the edge

1.2 Challenges in developing applications at the edge

In this section, we will further discuss and quantify the challenges in developing applications at the edge. There are two overarching factors that make development hard. The first is heterogeneity at the edge. The second is emerging application trends.

1.2.1 Heterogeneity at the edge

In this section, we will discuss heterogeneous edge environment, including hardware, programming environment and application requirement heterogeneity.

Hardware heterogeneity. The hardware devices at the edge are heterogeneous, ranging from embedded sensors, wearable devices, and mobile phones to edge servers with a full spectrum of computation capability and architecture [17]. In terms of computation capability, for the same workload, the completion times for these devices can differ by orders of magnitude. For example, when running the same OpenCV [18] face recognition workload, it takes 2263.71 ms for a Samsung Galaxy Nexus Android smartphone to process a single image, while it only takes 197.77 ms for an Intel Core i7 with 3.6GHz, 4-core CPU Server to process the same workload. In addition, different hardware has different processor architectures, from ARM64 in a Raspberry pi to a high-end GPU server with NVIDIA GPU support. This further complicates the application development on these devices.

Programming environment heterogeneity. Running on top of heterogeneous hardware, the programming environments at the edge are incredibly diverse, ranging from operating systems to different programming platforms to libraries and tools. For operating systems, in addition to commonly-seen operating systems like Windows and Linux on servers and Android and IOS on mobile phones/tablets, there are various lightweight operating systems designed for embedded devices, including Raspbian [19], RISC OS [20], Ubuntu Core [21] and more. Furthermore, the same application functionality can be built on different platforms. For instance, platforms like Tensorflow [22], Mxnet [23] and PyTorch [24] provide similar features to ease the development of deep learning applications, but have entirely different APIs. For libraries and tools, building an application typically involves various third-party tools. Choosing a proper set of libraries alone requires a significant amount of domain knowledge.

Application requirement heterogeneity. Different applications can have different performance requirements. For example, for intrusion detection in the smart home scenarios, low latency is the main performance target [25]. While for workloads like person re-identification (e.g., spot a criminal within the video data captured by different surveil-

lance cameras), the results' accuracy becomes the main target. Further, even for the same workload, the performance requirements vary with deployment scenarios. Take an object recognition application [26] as an example; when run on a mobile phone, it might value processing time more than accuracy compared to the same workload running in a data center, given the interactive nature of mobile phones.

Together, these aspects outline a large design space to explore to find tradeoff points between hardware, programming environment, and application requirements.

1.2.2 Emerging application trends enlarge design space

In addition to the heterogeneous edge environment, a second contributor to scalable application development at the edge is the emerging application trends. These include diverse device/edge/cloud interactions, diverse data types and rates, and concurrent and continuous application execution.

Diverse device/edge/cloud interactions. There are several reasons why the interactions among devices, edge and cloud are highly dynamic. First, the availability of computation resources (e.g., CPU, memory, GPU) varies on the edge device during runtime due to multiple workloads competing for the same resource. For instance, [27] observes that over 50% of the requests could break the real-time response requirements when multiple workloads are racing for the communication channel for offloading. Second, in addition to running on a single device, applications running at the edge also can be run on a cluster of devices. It can be a cluster of embedded devices like an FPGA cluster or a cluster of servers or even a mix of several different types of edge devices.

Diverse data types and rates. Since its inception, edge scenarios have seen increasingly sophisticated analytics workloads operating on multimodal data such as both video and audio [28, 29]. The type, amount and rate of data vary. For example, the data processing rate can range from processing a single data frame ($\sim$ 1 MB) at a time to processing multiple data streams ($>$ 100 MB/min) continuously.

Concurrent and continuous application execution. Applications at the edge are becoming increasingly sophisticated, enhancing our interaction with the environment. Some scenarios might involve multiple concurrent applications or modules working together (Google

Tango [30], Gabriel [31] and DeepEye [32]), where each module is individually computationally intensive and requires offloading to the edge. Worse, these applications embody a vision of continuous operations, further straining the already limited resources on mobile devices.

Altogether, these application trends enlarge the design space and increase the development complexity significantly.

Discussion. Given all these challenges, we argue that there should be a framework to serve as the middleware layer in the current edge applications' software stack, enabling scalable application development at the edge. Specifically, this middleware layer should fulfill the following requirements.

- From the application layer's perspective, this middleware layer needs to provide novel system abstractions for the emerging application trends on the edge. Specifically, for a single workload, our middleware layer needs to separate data management from actual processing logic, dealing with a huge amount and various types of data generated at the edge. For multiple workloads running simultaneously, it needs to provide a resource management service to handle concurrent workload execution.

- From the OS and hardware layers' perspective, our system needs to mask heterogeneous edge devices and various programming environments on top of them, providing an abstract notion of available resources to the upper-level application layer.

1.3 Contributions

In this thesis, we present a novel framework to simplify application development and deployment over a continuum of edge to cloud, serving as a middleware layer to bridge the gap between the application and underlying OS layer's in the current software stack at the edge. Corresponding to the requirements we have summarized above, the framework consists of three pieces, *Crystal*, *Video-zilla* and *LinkShare*. These pieces connect different dimensions of design considerations, corresponding to the generic application development, data management and resource management respectively.

Specifically, *Crystal* masks hardware heterogeneity with abstract resource types through containerization and abstracts the application processing pipelines into generic flow graphs. Further, it supports a notion of degradable computing for application scenarios at the edge that are driven by multimodal sensory input. *Video-zilla* is a generic data management service between video query systems and a video store to organize video data at the edge. We propose a video data unit abstraction based on a notion of distance between objects in the video, quantifying the semantic similarity among video data. Last, considering concurrent application execution, *LinkShare* supports multi-application offloading with device-centric control, with a system-level scheduler service that wraps over the operating system scheduler.

In summary, this thesis makes the following contributions:

- We identify and quantify key challenges of developing applications at the edge, pointing out that the overarching issue of applications development at the edge is scalability.
- We sample the design space and connect different dimensions of design considerations, corresponding to the generic application development, data abstraction and resource management abstraction respectively.
- We build a novel framework to enable easy application development, scalable data processing and resource management.

1.4 Dissertation roadmap

This dissertation is organized as follows: Chapter 2 first explains the background of edge computing and applications running at the edge and then gives a broad overview of the existing efforts of boosting application development at the edge. Chapter 3 then presents a case study further to illustrate the connection among our three pieces of work.

The following three chapters introduce the system abstractions and systems we build to enable scalable application development at the edge. Chapter 4 explains Crystal, a framework to simplify multimodal analytics development over heterogeneous edge scenarios. Following that, Chapter 5 introduces a novel data abstraction to organize multimedia data

better. Furthermore, we build Video-zilla, an indexing layer for computing scalable live video analytics across the edge and cloud. Chapter 6 focuses on multi-application execution at the edge and proposes a novel resource management abstraction. Based on that, we build LinkShare to enable Device-Centric Control for concurrent and continuous Mobile-Edge/Cloud Interactions. In addition, Chapter 7 discusses the challenges and possible solutions to organize heterogeneous and decentralized edge infrastructures.

In the end, Chapter 8 concludes this thesis and discusses a few future research directions extended from this thesis.

Chapter 2

Background

2.1 Canonical applications

There are various applications running at the edge. We explain these applications and highlight their development requirements.

Smart video surveillance. Recent years have witnessed a sharp uptick in the number of video cameras. 2018 alone saw 140 million shipments of video cameras [33]. According to the British Security Industry Authority [34], approximately 300,000 cameras are already deployed in UK schools. The boom of video cameras deployment, especially from mobile phones and low-cost IP surveillance cameras has led to an exponential growth of video data.

Due to the limited maximum attention span of any personal monitoring (around 20 min), there is a drive to automatically analyze these video data [35] This led to the emergence of smart video surveillance systems [36–38]. The computation cost of video processing is high. To ensure low latency and efficient network bandwidth usage, edge computing has been seen as a promising approach [39].

Multiple queries can be run on a smart video surveillance system. One classic example is an intrusion or crime detection application [40]. A video surveillance system is expected to provide a real-time alert when a situation of interest occurs, for instance, when an unknown person breaks into a house. Another commonly seen workload is traffic monitoring. Works like Vision Zero [41] aim at reducing traffic accidents by detecting "close-calls" between cars, bikes, and pedestrians. This detection query helps preemptively deploy safety measures.

In addition to these two types of workloads, many more queries can run on a surveillance system, including vehicle counting for congestion control [42], license plates recognition for suspicious car detection [43] and poorly-visible pedestrians detection (for instance, blocked by other cars) to aid decision in autonomous-driving cars [44].

Industrial IoT. Sometimes termed "Industry 4.0", the fourth industrial revolution aims to realize interconnected, intelligent manufacturing systems by integrating advanced manufacturing techniques with the Industrial Internet of Things (IIoT) [45]. Billions of computation and sensory devices will be deployed to perform various tasks. These devices monitor diverse conditions, such as temperature, humidity, and anomalous audio. Given the volume and sensitivity of the data, on-premise (private-cloud) data analytics is preferable to public cloud offloading [46]. Both IBM [47] and SIEMENS [48] provide edge-based products for manufacturing quality assurance.

Running in a private edge cluster, a smart manufacturing system like that in [49] typically needs to support various workloads. For instance, a data-driven risk assessment for industrial manufacturing systems has been presented based on real-time data [50]. Another example is fault detection of sensors in a manufacturing scenario. [51] presented real-time and robust sensor fault detection in a hydraulic system. In addition to these workloads, there are others like automatic assembly line resequencing [52], supply chain optimization [53] and interactive industrial robots [54].

Wearable assistance. Leveraging sensor data to provide cognitive assistance has long been an active area of research for IoT applications and commercial products, e.g., Google Lens [55] using images from mobile/wearable cameras and human activity recognition using audio input and inertial unit sensing data on earphones [56, 57]. These often run some processing on the device to provide real-time response.

For instance, Gabriel [31] provides interactive cognitive assistance using Google Glass to help people suffering from cognitive declines, such as those with Alzheimer's disease. The patients are often unable to remember the names of friends or remember to perform daily tasks. When looking at a person that the user might know, the assistant will tell the user the name of the person immediately. When looking at his/her plants, it will remind him/her to water them. These two scenarios require face recognition and object recognition

respectively.

Discussion. The wide range of scenarios outlined above share several common development challenges:

First, all these workloads run on heterogeneous edge devices with mixed processing capabilities (CPUs, GPUs, FPGAs). Even for a single workload, running on different devices can have orders of magnitude performance difference [26].

Second, a single workload may need to process a massive amount of data, and even more than one input stream. For instance, a surveillance camera capturing 1080p video at 30 frames per second captures 200 GB of video per day. A crime detection program that needs to run over 100 such cameras will need to process 20 TB of data per day. Another example is the interactive industrial robots in IIoT. A robot can take audio, video and more input from various sensors in it to perform workloads like path finding [54].

Third, multiple workloads can run simultaneously on the same set of edge devices. Further, one workload may need to process data generated by multiple IoT devices. Take the sensors' fault detection in the industrial IoT scenario as an example; here one needs to jointly consider data collected by all the available sensors to detect the failed sensors in a system.

Despite all these common challenges, the state of the art is to deploy expensive custom solutions with application-specific monolithic software stacks at the edge, demanding significant amounts of domain-specific knowledge and repeated development effort.

2.2 Related works

In this section, we will review several directions of research carried out in edge computing. We only discuss general edge references are listed here. Additional work more specifically related to individual projects will be covered in the following chapters.

2.2.1 Different edge concepts

There are several different terms when one refers to edge computing. In this section, we carefully discuss these terms, and these terms are interchangeably used throughout this

thesis.

Cyber forging. The origin of the idea of edge computing can date back to two decades ago. [58] proposes the concept of *cyber forging*, which is a mechanism to use "opportunistically discovered servers in the environment" to improve the performance of applications on mobile devices. Mobile devices can offload their workloads to nearby servers called surrogates. These surrogates can be located in public spaces like airports and coffee shops. Fifteen years later, [59] provides a valuable reflection on *cyber forging*, summarizing the main technique achievements and remaining challenges.

Cloudlet. The concept of *cloudlet* was first introduced in [60] and a prototype implementation at CMU. As its name suggests, a *cloudlet* is a small-scale cloud datacenter that is located at the Internet edge. Serving as a middle tier, it bridges the mobile devices and cloud. Compared to the concept of *cyber forging*, a *cloudlet* takes a step forward to consider the impact of the cloud as well.

Mobile edge computing. The term *mobile edge computing* was standardized by European Telecommunications Standards Institute (ETSI) and Industry Specification Group (ISG) [61]. According to ETSI, *mobile edge computing* is defined as: "Mobile edge computing provides an IT service environment and cloud computing capabilities at the edge of the mobile network, within the radio access network (RAN) and in close proximity to mobile subscribers." As we can see from this definition, mobile edge computing weighs more on the integration between computation capabilities and the radio access network.

Fog computing. Along with the emerging Internet of Things, fog computing was first proposed in [62]. "Fog" originates from its cloud-like computation properties, while it is closer to the "ground", i.e., the Internet-of-Things devices.

2.2.2 Enabling techniques

Application offloading. Computation offload to the cloud has been explored extensively in the last 20 years [63–70]. Several works (for instance, MAUI [12], Thinkair [15], COMET [14], CloneCloud [13] and more [16, 71]) focus on seamlessly partitioning a single workload spanning the mobile device and the remote server. Further, works like MCDNN [72] provide a common platform for multiple applications concurrently utilizing deep neural net-

works (DNN).

Workload placement. Partitioning and placing a workload across end devices, edge devices and potentially cloud servers is the key enabler of edge computing. Workload placement has been extensively discussed in the literature [73–75]. Typically, workload placement involves finding the available resources that satisfy the applications' requirements and optimizing performance objectives. Existing works can classified by the performance goals they target, including latency [76–78], resource utilization [79–81], energy consumption [82, 83], monetary cost [84, 85] and more.

Operating system virtualization. Operating system virtualization allows multiple operating systems to run on a single physical machine. Like cloud computing, edge computing is highly dependent on system virtualization that decouples hardware from upper-level software in order to support features like multi-tenancy [86–88]. However, as edge computing differs from cloud computing in terms of constrained resources and heterogeneous devices, heavyweight virtualization techniques like the virtual machine [89] cannot be applied to edge computing. These constraints demand the deployment of lightweight virtualization techniques like containers [90] and unikernels [91]. Based on application requirements and resource availability, these techniques may also need to be used in combination [92].

2.2.3 Emerging edge infrastructures and testbeds

This section explores emerging edge infrastructures from both the hardware and software perspectives. Furthermore, we will discuss some recent testbeds that are developed to help test edge computing platforms.

Edge infrastructures. An edge infrastructure is composed of hardware and software to manage the computation, storage and network resources at the edge. From the hardware perspective, a wide range of devices including drones, network gateways, WiFi Access Points (APs) and edge ISP servers can serve as computer servers for efficient resource provisioning [93]. For instance, FocusStack [94] uses multiple Raspberry Pi boards installed in connected vehicles to build a video-sharing application at the edge. [95] aimed at building an edge infrastructure with laptops and mobile phones used in public environments like movie theaters and cafes. Another option is co-locating computation devices with network gate-

ways. Works like GENI [96] integrate network, computation and storage resources together to collect rich datasets for public safety surveillance and vehicle internal-state sensing and modeling at the edge.

From the software perspective, in addition to the operating system virtualization we mentioned above, another important pillar is the network resource virtualization. An edge infrastructure can adopt software-defined networking (SDN) and network functions virtualization (NFV) for managing the network through software [97]. Built on top of SDN/NFV, all virtual computation units for the same tenant can be in the same virtual LAN (VLAN), even if they are located in different areas.

In addition, works like [98–100] tried to explore the middleware layer for application development at the edge. However, they restricted themselves to a limited set of mobile devices. For example, [99] mainly aims at Android smartphones, which can not be easily generalized to the wide range of emerging edge devices.

Edge testbeds. Along with the emergence of various edge infrastructures, there are efforts from both academia and industry to provide testbeds to boost edge research. For instance, initially designed for testing ideas on a configurable cloud platform [101], the Chameleon testbed is evolving to include a preview set of new capabilities to allow users to experiment with edge devices [102]. Another effort made in academia is the 5th generation test network (5GTN) architecture [103]. 5GTN allows third-party edge service providers to test their applications in a 5G mobile edge computing framework. The Platforms for Advanced Wireless Research (PAWR) program [104] funds several more testbeds [105–108], aiming at "enabling the emerging Internet of Things(IoT), edge computing and heterogeneous wireless connectivity technologies."

In addition, there are also several testbeds from industry. For instance, Nokia and its partners delivered an intelligent car-to-car infrastructure communication system over LTE network [109], opened an opportunity for connected-vehicle application developers to test their ideas.

2.2.4 Security and privacy

On the one hand, edge computing has provided significant assistance for efficiently processing huge workloads created by various IoT devices. On the other hand, its hasty development has led to potential security and privacy concerns. In this section, we explore the existing efforts that are dealing with security and privacy issues at the edge.

It is important to implement security and privacy mechanisms to safeguard the edge resources from any intrusion. Here, we mainly discuss the two most important perspectives: 1) identification and authentication and 2) data security. For more detail, papers like [110,111] provide a thorough survey of security and privacy threats and defense mechanisms at the edge.

Identification and authentication. In cloud computing, a data center is typically owned by a single cloud service provider. However, in the edge paradigm, a group of edge devices can be hosted by several providers. Proper identification and authentication are required to enable everyone in the ecosystem, such as end devices, edge service providers, and even cloud service providers, to authenticate each other mutually. User-friendly solutions, including authenticated key exchange within local wireless ad-hoc networks [112] or via NFC [113], have been proposed to provide a secure authentication service. For authentication between users and the cloud when the edge is temporarily unavailable, [114] provides a stand-alone authentication mechanism to enable user devices with the corresponding cloud servers.

Data security and privacy. In an edge computing paradigm, user data can be outsourced to edge servers. This leads to several potential security problems. For example, outsourced data can be modified or even deleted because of malicious computation activities at the edge. There are several efforts to mitigate this concern. Similar to that in the cloud, some apply [115–119] appropriate methods to audit data storage at the edge to confirm that data are properly stored. In [120], a verifiable computing protocol is proposed to provide a computationally-sound proof. In addition to the works mentioned above, data encryption is another crucial way to protect data security. Works like [121–123] provide various encryption mechanisms to deal with different data usage scenarios.

2.2.5 Discussion: missing points

Unfortunately, existing works cannot scale application development at the edge. The overarching issue is that application developers still need to make hard decisions, despite all these tools, when dealing with each of these issues. This leads to application-specific solutions. Because of that, application development at the edge still needs a significant amount of domain knowledge (e.g., the specifics of the application and the execution environment setup). For instance, a video analytic application can run on both a smartphone and an IP camera integrated with a Raspberry pi. As an application developer for now, one still needs to manually figure out the hardware specs and the runtime environment setup to implement the application-specific logic.

Further, because certain issues (like the need to deal with heterogeneous hardware we mentioned in the example above) apply across applications, application developers tend to make repeated efforts when developing these applications. Therefore, in this thesis, we propose a novel framework to simplify application development and deployment over a continuum of edge to cloud, providing end-to-end support to application developers.

Chapter 3

Case study: Smart video surveillance applications at the edge

In this chapter, we present a case study to further illustrate the connection among the three pieces of work in this thesis.

3.1 A benchmark application

We show a typical development process of a smart video surveillance application at the edge, on top of a multi-camera system with 12 video surveillance cameras.

The main reason we chose video analytics is that it is the killer app at the edge. Specifically, we implement an object re-identification application to associate a particular object across different scenes and camera views. An exploratory object re-identification query can be "find a car in video data captured by those 12 video cameras within the last hour." In the runtime, users can send their queries to servers in the cloud and get the query results. Each surveillance camera can offload the captured video to a nearby edge server and, optionally, to the centralized data center.

DNN-based visual feature matching is the state-of-the-art solution to the object re-identification problem. In our case, we choose to use ResNet50 (pretrained on Microsoft

COCO dataset [124]) as the feature extractor for our re-identification problem. For comparison, we build our application both directly on the raw video data and on Video-zilla, a video indexing layer that groups semantically similar video data together. Video-zilla uses an ingestion feature extractor to help extract semantic information from data. We use VGG16 (pretrained on Microsoft COCO dataset), a lightweight object feature extractor to serve as the ingestion feature extractor.

Various deployment scenarios. We consider two different edge-cloud deployment scenarios. We start with a simple homogeneous deployment scenario where all edge servers have the same hardware specification.

However, edge devices, in reality, are heterogeneous [10]. Both the application itself and our Video-zilla indexing system can be run on a range of devices with various processing capabilities. To automatically deal with this heterogeneity, we encapsulate both the application and Video-zilla index system to run on top of Crystal. Video-zilla is abstracted as a flow graph in Crystal, represented by a set of interconnected containers. In our current implementation, our benchmark application is built by extending the query interface processing component provided by Video-zilla. When running our benchmark application, essentially, we are running a customized version of Video-zilla with application-specific logic.

Real-world dataset. we assemble publicly available real-world datasets to emulate a real-world multi-camera surveillance system for public transportation, like that in Chicago [125]. The total length of these videos is 8 hours, including three types of feeds from both stationary and moving cameras: i) 10 road-view captured by in-vehicle cameras: 5 of them capture the downtown areas [126] of New York City, London, Chicago and Los Angeles, one per city; the other five capture highways across the U.S. [127]; ii) 1 train-station video live streams from Youtube [128]; and iii) 1 harbor video feeds from Youtube [129]. We intentionally set the number of cameras in train stations and harbors to be smaller than that of in-vehicle cameras, which matches the expected ratio between these feeds in practice.

Hardware setting. For the homogeneous setting, we use two Linux servers to host the hierarchical index, both with 8-core 2.1 GHz Intel Xeon CPUs, one with a NVIDIA RTX 2080Ti GPU, and the other with an NVIDIA RTX 2070 GPU. We run an inter-camera index on the former and intra-camera indices on the latter. For the heterogeneous setting,

we keep the server with an NVIDIA RTX 2080Ti GPU to run an inter-camera index. For the intra-camera indices, we run them on three different devices: a GPU server with an NVIDIA 2070Ti GPU, a CPU server with a quad-core Intel Core i7 6700 3.4 GHz CPU with 8 GB memory, and a Raspberry Pi 4 with 8 GB memory. These devices represent typical processing units across the edge and the cloud, covering a wide range of heterogeneity.

3.2 Quantitative results

We run the object identification application in both the homogeneous and heterogeneous settings. Specifically, the query is in the format of *find image frames that contain object X within all video data.* We let X be a fire hydrant, boat or train, which is represented by corresponding image frames that contain this object. We generate 10 query instances, taken from the real-world videos, per query type.

Figure 3.1 shows the normalized accumulated GPU time when processing these queries with and without Videozilla. Compared to running queries directly on the raw video data, Video-zilla uses 50 times less GPU time for the same set of queries at the cost of less than 3% accuracy loss.

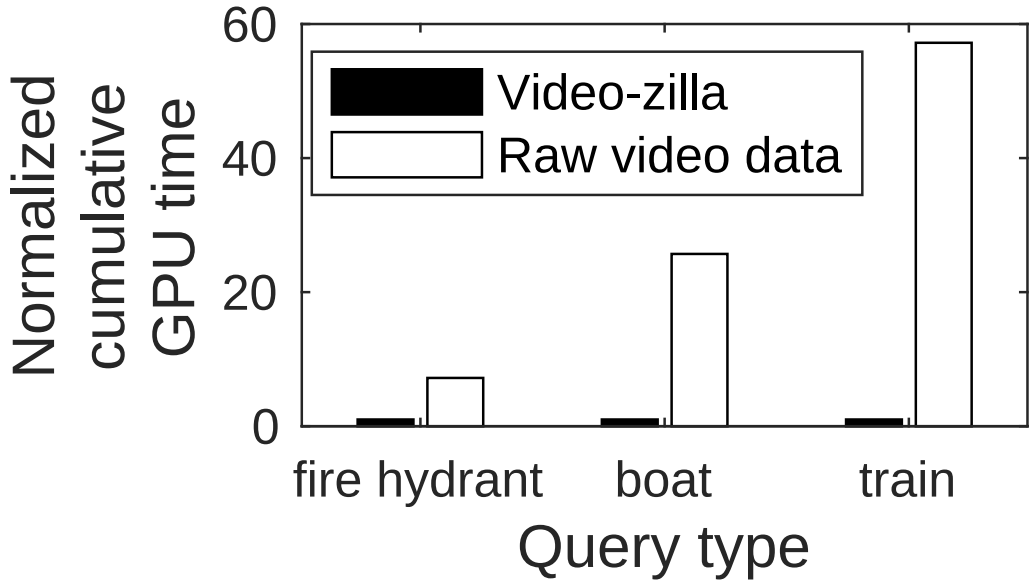


Figure 3.1: Total GPU time

When considering the heterogeneous deployment scenarios, due to the operating system layer's difference, migrating our benchmark application from a GPU server to a CPU server and Raspberry pi boards requires a signification amount of development effort. As mentioned above, we encapsulate both the application and Video-zilla index system to run on top of Crystal. The original Video-zilla implementation totals about 4K lines of code, most in Java. With the help of containerization support provided by Crystal, it took less than 100 lines of code to port Video-zilla to all three different devices. Furthermore, thanks to the built-in featurizer library in Crystal and the query API provided by Video-zilla, defining our benchmark application to work on these three devices takes less than 30 lines of code. Listing 3.1 shows the pseudo-code snippet.

```
public Module[][] build() {
    // Crystal APIs to specify feature extrator
    Module resultRefinement = new Module("video", "ResNet50");

    // Video-zilla APIs to specify query logic, pass a function
       handler
    Method queryResult = new Method(getResults);
    affectRec.setConfig("config.yaml");
    affectRec.input("path/to/object/of/interest")

    // Crystal APIs to link the above two modules
    Module[][] app = {{queryReuslt, resultRefinement}};
    return app;
}

public List<INDArray> getResults(INDArray objectImg) {
    // Video-zilla query APIs
    List<INDArray> results = directQuery(objectImg, "object
       identification");
    return results;
}
```

```
// deviceType can be "cpu", "gpu" and "raspberry4".
public void run(device, deviceType) {
    Module[][] app = build();
    this.setConstraint(app, "accuracyLoss", 10, "lessThan");
    this.setGoal(app, "time",  "minimize");
    this.setGolden("path/to/ground-truth/results");
    // actually start the execution.
    this.execute(app, device, deviceType);
}
```

Listing 3.1: Building an example Crystal Application

3.3 Beyond a single application

So far, we have focused on building a single application at the edge in this case study. However, as we mentioned above, multiple applications can run on the same set of edge devices. When it comes to the video analytic scenarios, various applications, including moving object tracking [130], object identification [131], face recognition [132] and more, can be hosted by a single smart video surveillance system. These workloads compete not only for computation resources at the edge but also for network resources. To deal with network resource contention across these workloads, we built LinkShare to enable device-centric network control for concurrent and continuous Mobile-Edge/Cloud Interactions.

Chapter 4

Crystal: Simplifying multimodal analytics development over heterogeneous edge scenarios

4.1 Introduction

Since its inception, edge scenarios have seen increasing complexity in terms of more diverse execution settings [57, 133] and more sophisticated analytics workloads operating on multimodal data such as both video and audio [28, 29]. The notion of "edge" is constantly broadened to include the full spectrum of processing capabilities, ranging from wearable or embedded devices to a small server cluster in or near an enterprise. This exacerbates the hardware heterogeneity already seen in edge *devices* [17, 134] to disparity over orders of magnitude. Meanwhile, multimodal analytics is substantially more complex than its single-modal counterpart, due to added data processing pipelines and more inference options. (Section 4.2).

Existing multimodal, edge analytics applications tend to be monolithic and purpose-built, ranging from research prototypes intended to explore new applications [135–137] or extract more insight from the multi-modal data [138], to industry solutions to individual use cases (e.g., Siemens Industrial Edge for Industrial IoT [48]. Their monolithic nature

mixes generic application logic with specific execution and deployment considerations. Not only does this preclude code reuse, the development process requires substantial domain knowledge to implement multimodal learning, optimize for diverse hardware processing capability, and manage dependencies between the toolchains.

To address these issues, we design and implement Crystal, a framework to ease multimodal analytics application development over diverse edge settings. Building on Crystal, the developer only needs to write tens of lines of application code regardless of the execution setting, and the framework then performs necessary adaptations to the desirable deployment settings and performance targets (Section 4.3).

Crystal refactors the application processing into three abstract layers (Figure 4.1): *application flow assembly*, *execution shaper*, and *containerization*. Containerization masks hardware heterogeneity with abstract resource types (Section 4.4), while the assembly layer abstracts the application processing pipelines into generic flow graphs as well as provides built-in library support for common analytics modules (Section 4.5). The middle shaper layer then supports a notion of degradable computing to map the application flow to the available resource and streamlines the analytics processing (Section 4.6). This architecture decouples application-specific logic, generic optimization strategies, and hardware-specific deployment considerations from one another.

Beyond code reuse and facilitating application development with minimal domain knowledge, Crystal provides system support to streamline the processing flow and adapt to hardware capability that may vary by orders of magnitude. The application assembly layer generates alternate flow graphs by skipping the processing of one or more modalities, based on implicit data redundancy across modalities (Section 4.5). This expands the potential optimization space to cater to the level of hardware disparity. The shaper layer introduces a notion of *approximate Pareto-frontier*, i.e., a small subset of the optimization space that identifies the performance boundaries, and then navigates the search space efficiently through a combination of offline and online profiling to quickly identify the application configuration settings to match the hardware processing capability (Section 4.6). Further, Crystal provides wrappers and lightweight data passing mechanisms (Section 4.7) to facilitate interfacing with existing toolkits such as open-source pre-processing packages in

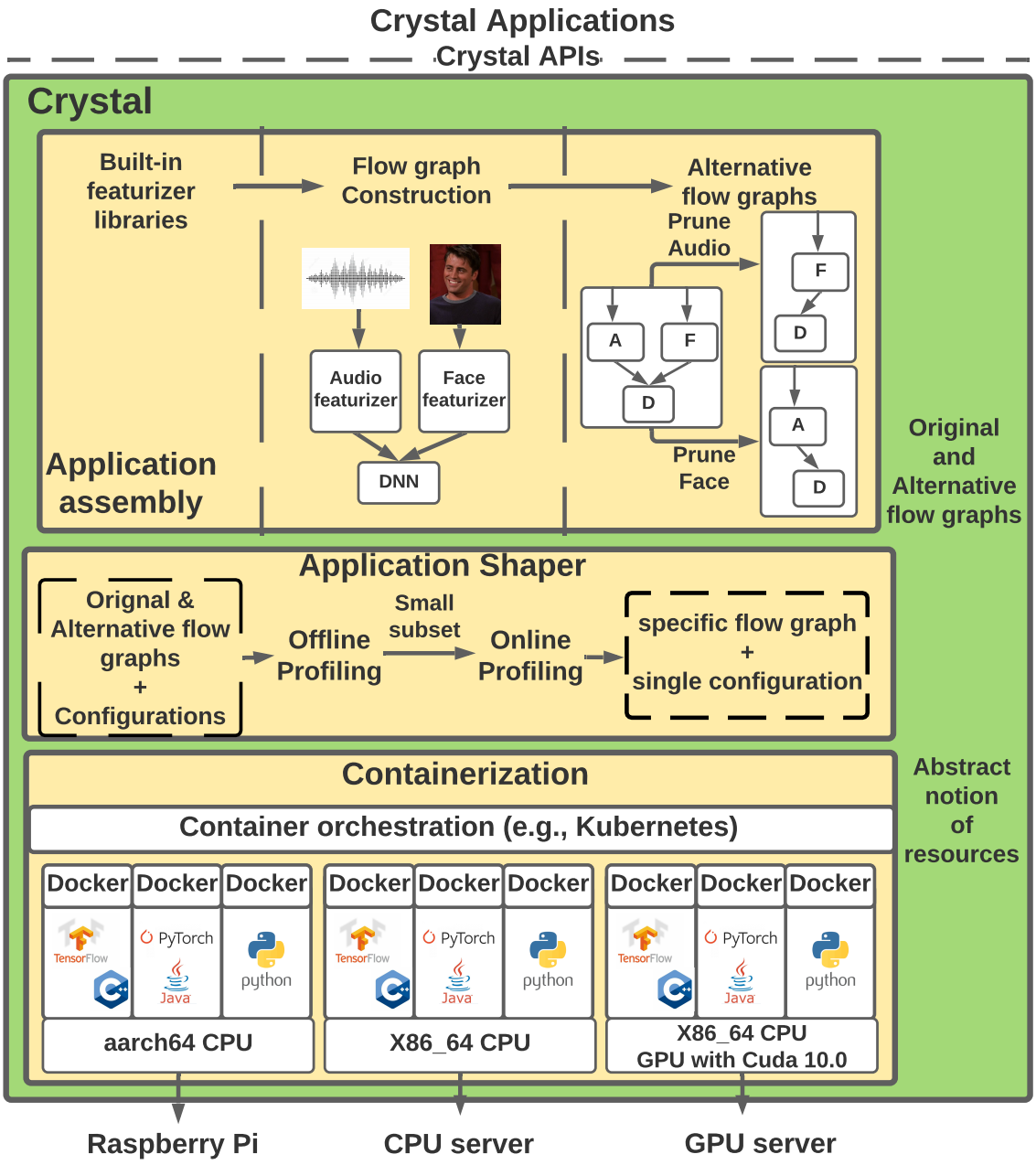


Figure 4.1: Crystal Architecture

different languages (e.g., Matlab for audio [139], C++ for face recognition [140]), machine learning or deep learning frameworks (e.g., TensorFlow and PyTorch), and container technologies (Docker and Kubernetes).

Extensive evaluation (Section 4.8) shows that Crystal reduces the application development effort to writing only tens of lines of code for the application logic or adding new processing modules. Crystal can degrade or scale execution gracefully across a continuum of processing capability, from resource-constrained settings like a Raspberry Pi to a small EC2 server cluster, while operating within the desirable performance targets.

4.2 The need for framework support

Consider the following three application domains:

Wearable assistance. Leveraging sensor data to provide cognitive assistance has long been an active area of research for IoT applications and commercial products, e.g., Google Lens [55] using images from mobile/wearable cameras and human activity recognition using audio input and inertial unit sensing data on earphones [56, 57]. These often run some processing on device to provide real-time response.

Smart spaces. Home and office automation has been studied since the early days of the IoT vision [141]. These may draw on simple measurements like temperature and humidity and/or security camera feeds [142], driven by gestures, facial expressions, or voice commands [143]. While existing home assistants like Alexa are cloud-based, there is also interest in confining sensitive personal data within the home via an in-home server or controller[1].

Industrial IoT. Sometimes termed "Industry 4.0", the Industrial Internet of Things (IIoT) refers to interconnected sensors and other devices used to drive manufacturing automation [144]. Sensors monitor diverse conditions, such as temperature, humidity, and anomalous audio. Given the volume and sensitivity of the data, on-premise (private-cloud) data analytics is preferable to public cloud offloading [46]. Both IBM [47] and SIEMENS [48] provide edge-based products for manufacturing quality assurance.

1. ISPs like at&t and Comcast are interested in adding compute service to their existing bundles.

4.2.1 Heterogeneity in Edge Scenarios

The three examples above highlight vastly different "edge scenarios". The mode of processing varies from single application on a single device, to potentially multiple applications on a server cluster. There are two immediate consequences of the hardware heterogeneity, i.e., diversity in the development and execution environments, and disparity in the processing capability.

Running a single application on various types of devices itself is non-trivial, due to the dependencies in the development and execution toolchains. Consider a simple image featurizer, using PyTorch-based VGG19_BN [145]. Running this on a Raspberry Pi 3 (32-bit CPU) vs Pi 4 (64-bit CPU) requires different pre-built PyTorch and resolving operating system dependencies given different instruction sets.

Worse, the device processing capabilities range from a low-power processor on an embedded device, to multi-core CPUs or GPUs on a single server, to multiple powerful servers. This then translates to wide-ranging performance for the same processing task. It takes 14.75, 246.42, and 66040 seconds to run VGG19_BN over 1000 images on an Nvidia 2080Ti GPU server, an Intel Core i7 CPU server and a Raspberry pi 4 respectively, i.e., orders of magnitude performance difference!

4.2.2 The Complexity of Multimodal Analytics

While multi-modal *learning* dates back decades [146], what is new is that multimodal analytics applications are becoming sophisticated, more diverse, and maturing from research prototypes focusing on data mining to real-world applications [147]. Consider multimodal affective recognition application, MFN, as an example. MFN takes three modalities, video, audio and text, as inputs to recognize the affection of the person of interest. Figure 4.2 illustrates the processing pipeline of MFN.

Compared to single-modal analytics, incorporating additional input modalities incurs complexity and challenges.

Multiple data types. For example, text and video data differ in the data formats, but more importantly the feature extraction mechanisms (i.e., the *featurizers*). Across modali-

ties there is a range of open-source toolkits and proprietary solutions, varying in maturity, but more importantly, different language implementations. For example, audio processing packages are often in Matlab to leverage the built-in signal processing toolboxes, while popular computer vision libraries are variously implemented in Python. Since multimodal learning is an active research topic, new models are proposed frequently, impeding standardized implementations.

Multiple processing branches. Since each modality needs specific processing steps, a multimodal application pipeline starts with multiple separate branches and merge them at some point. This makes the overall performance susceptible to potential imbalance in the input data rate and processing rate between the branches. A delay in any branch can potentially bottleneck the overall application processing.

Multiple inference paradigms. Depending on whether the individual modalities are pre-processed before being fed into a final inference model, multimodal applications can adopt *late fusion*, *early fusion*, or *mixed fusion*. This means the processing pipelines of multimodal applications can take on several shapes.

From single-modal to multi-modal. Extending single-modal analytics to multimodal analytics introduces new challenges and optimization opportunities.

First, the multiple branches in a multimodal application requires "cross-branch" support to streamline the overall application flow. This includes support for lightweight data passing and multiple language runtimes, if the modality-specific featurizers are implemented in different languages. In contrast, single-modal applications can be abstracted into model serving workloads [148], and containerization alone, using existing technologies, is roughly sufficient to deploy these on heterogeneous devices.

Second, multiple modalities introduce additional dimensions to the performance optimization space, due to the inherent redundancy across different modalities [147, 149, 150]. Thus adding or removing one or more modalities introduces extra performance tradeoffs. Back to the MFN example, the three input modalities essentially contain part of the state or the person of interest. The spoken language information is implicitly covered by both audio and text data.

Third, performance optimization involves more than tuning the final inference model.

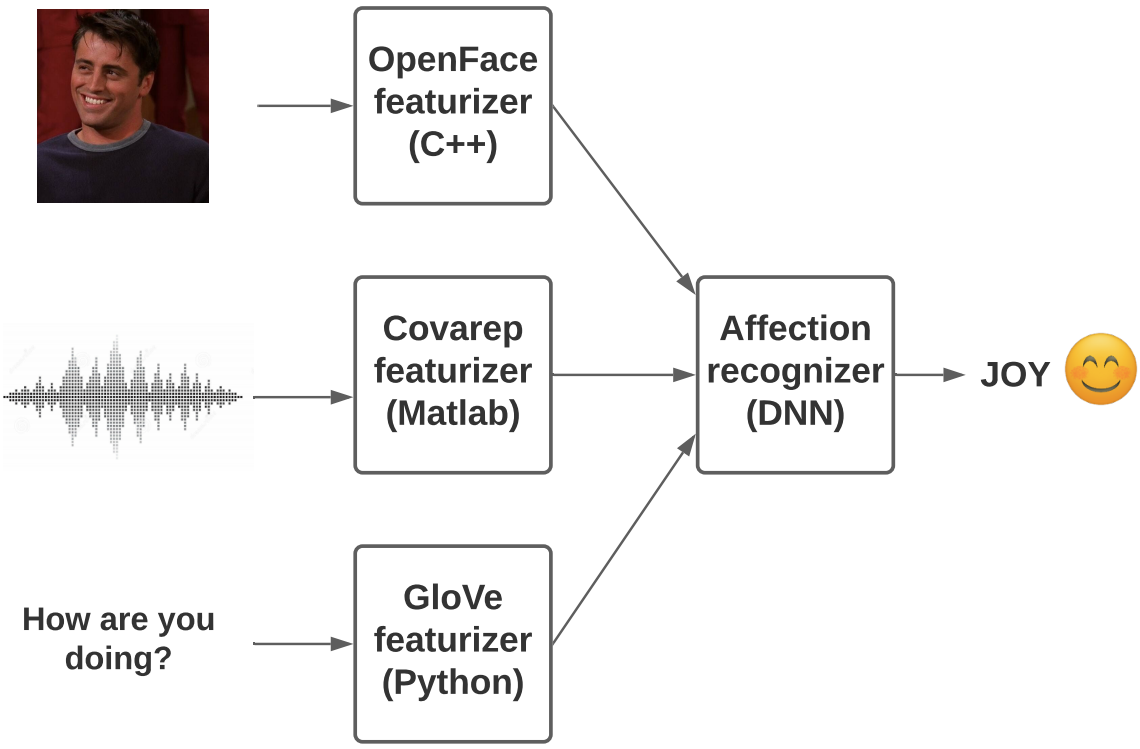


Figure 4.2: MFN: multimodal affective recognition

The inference performance of a single-modal application is bound by the inference model, and adapting this model [134, 151] only is often sufficient to explore tradeoffs between inference accuracy, latency, and resource availability. In contrast, the main performance bottleneck of a multimodal application may instead be a preprocessing step. In the MFN example, the audio featurizer, not the final affective recognizer, contributes to the bulk of the overall processing time.

4.2.3 Existing approaches and limitations

Existing ML/DL application implementation frameworks [148, 152] are limited in several ways.

First, they are typically designed for cloud-based deployment scenarios, where the dynamic workload pattern instead of the hardware heterogeneity is the main challenge. They do not provide built-in support to resolve toolchain dependencies. Second, the mechanisms provided are inadequate to handle orders of magnitude difference in the resource availability.

Initially designed for single-modal analytics, these frameworks do not provide mechanisms to handle multi-modal analytics effectively. For example, without leveraging the redundancy across different modalities, they miss a wide range of performance trade-off possibilities. Third, these frameworks are designed for single-language runtimes, while multimodal analytics raise the possibilities of using existing packages implemented in different languages. Using existing frameworks requires porting necessary packages to the same language, which can be tedious and error-prone.

Our approach. To address the above challenges and fill in the gap between existing solutions, we design and implement Crystal to provide generic system support for multimodal analytics across a continuum of edge scenarios. We describe the overall architecture and the APIs next, followed by detailed descriptions in subsequent subsections.

4.3 Crystal Overview

Crystal comprises three layers (Figure 4.1): application flow assembly, application shaper, and containerization. The three-layer architecture is inspired by Sapphire [153], separating application logic from deployment. However, we need to add the shaper stage to further tailor an application to heterogeneous processing capabilities.

Containerization. The lowest container layer directly interfaces with heterogeneous hardware units, regardless of whether the execution is intended for a single device (a raspberry pi, a CPU or a GPU server) or a server cluster (CPU/GPU clusters), to provide an abstract notion of resources to the layer above. This layer essentially provides runtime instances with wrappers around the basic Docker containers to include dependencies. Each type of hardware is encapsulated in a number of container instances, each instance set up with the runtime dependencies for specific combinations of machine learning frameworks (e.g., TensorFlow and PyTorch) and language bindings (e.g., for Java, C++, or Python). Each processing component in an application pipeline runs in its own container. This layer can interface with a container orchestration framework like Kubernetes [154] to allow the application to run across multiple devices. Running on a single device is equivalent to running on a single-node cluster. (Section 4.4)

Application assembly. The top layer in Crystal interfaces with the analytics application. It provides built-in library support to the application developer to simplify application pipeline construction, and combines these and the custom application logic to assemble the processing flow in terms of a directed acyclic graph (DAG). Further, Crystal generates alternate DAGs by pruning one or more modalities. These DAGs are presented to the lower shaper layer. (Section 4.5)

Shaper. The shaper aims to "adapt" the abstract flow graphs to fit to available compute resources and map the graph components to the containers. This layer selects one of the DAGs from the assembly layer, as well as suitable configuration settings for each component on the DAG. Crystal first conducts offline profiling to reduce the search space and then an online profiling phase to finalize the actual flow graph and configurations. (Section 4.6)

4.3.1 Crystal APIs

Crystal exposes four types of APIs to application developers:

Processing hardware specification. The target execution device can be specified by its IP address and device type. Crystal currently supports three types of hardware, "2080 Ti GPU server," "Intel Core i7 CPU server," and "Raspberry Pi 4."

Defining processing components. Each processing step in an application flow graph can use either a built-in method or a new user-defined mechanism. The former can be specified with `Module()`, while the latter can be invoked via `Method()`, which is just a simple wrapper around a user-defined function. To better prepare for shaping, developers also need to specify the output data type of the custom method using textttsetOut(), as well as any user-defined performance metric and calculation method using `setCalMetric()`. The input data to the inference process can be either read from a file on the disk or streaming data from a specified network port, using `setSource()`. All data for the same modality are merged into a single stream.

For applications that are based on deep learning inference, Crystal interfaces with an existing automatic model compression tool. An application developer can specify the deep learning model file containing both the parameters and structural information that they use in the customized inference module by calling `Model()` API. In the meantime, applica-

tion developers must provide the training/validation/test datasets by calling `setTrain()`, `setValid()` and `setTest()`.

Execution pipeline specification. After defining individual components in a flow graph, specifying the flow graph itself is straightforward. The actual graph is represented as a 2-D Module array, and each pair of Modules represents a source-destination edge in the flow graph.

```
public Module[][] build()
    Module videoFeaturizer = new Module("video", "VGG16");
    videoFeaturizer.setSource("/path/to/video/input/file");
    Module audioFeaturizer = new Module("audio", "Spectral_Rolloff")
        ;
    audioFeaturizer.setSource("localhost:5600");
    Model lstm = new Model("path/to/model");
    lstm.setTrain("path/to/train/dataset");
    lstm.setTest("path/to/test/dataset");
    lstm.setValid("path/to/valid/dataset");
    Method affectRec = new Method("path/to/algorithm/execution/logic
        ", lstm);
    affectRec.setOut("affection", int);
    affectRec.setCalMetric("accuracy", "calculateF1");
    affectRec.setConfig("config.yaml");
    Module[][] app = {{videoFeaturizer, affectRec}, {audioFeaturizer
        , affectRec}};
    return app;
public void run(device, deviceType)
    Module[][] app = build();
    this.setConstraint(app, "accuracyLoss", 10, "lessThan");
    this.setGoal(app, "time",  "minimize");
    this.setGolden("path/to/ground-truth/results");
    this.execute(app, device, deviceType);
```

Listing 4.1: Building an example Crystal Application

4.4 Masking heterogeneity

Crystal masks the diversity of execution environments across heterogeneous devices through a container layer. When developers deploy an application on device, there is one container for each processing component, such as an audio featurizer, within the application. Crystal hooks to Docker [155] to provide container support.

Individual container formation. Each container in Crystal combines all sorts of bindings to run a single processing component on a specific device. Crystal supports language specific bindings, including C++, Java (supporting Maven and Gradle dependency managers), Python (supporting pip dependency manager). On top of that, for multimodal applications in particular, one application may include one or more DNN-based modules. Crystal supports the bindings for two widely-used DNN programming frameworks, Tensorflow and Pytorch. For the same set of bindings, such as TensorFlow with an additional C++ bindings, Crystal includes two versions to deal with X86_64 and aarch64 CPU architecture respectively. Besides, Crystal also supports NVIDIA GPUs differentiated by various NVIDIA Cuda versions. Currently, Crystal supports bindings for NVIDIA Cuda 9.1, 10.0 and 10.1, which cover a wide range of NVIDIA GPUs.

Container orchestration. When running on more than one device, existing container orchestration frameworks can balance load, auto-scale, and detect and handle failure automatically. Crystal integrates with Kubernetes [154] to perform all these functions. At run-time, Kubernetes runs as a background daemon, and Crystal submits requests to Kubernetes using `kubectl apply` to deploy applications on specific devices.

Other deployment setups. By separating the container layer from the upper layers, Crystal can be easily extended to support extra deployment requirements.

For instance, to support multi-tenancy, instead of Kubernetes, we can interface with the framework to manage multi-tenancy; it just needs to expose to the shaper layer the specific amount of resource allocated to each tenant, which might be dynamically adjusted.

For another example, consider supporting certain types of split edge-cloud execution. Unless the application pipeline (e.g., Neurosurgeon [156]) or the DNN model is split dynamically [157, 158], Crystal can also directly interface with such a runtime. This new

runtime provides containers corresponding to the cloud or edge execution; The developer can provide the respective application logic corresponding to the cloud and the edge (the same logic or adapted to each), which can then be mapped to the corresponding containers by Crystal.

4.4.1 Device registration

When deploying an application on a new device with already supported processors, application developers can call `registerDeviceType(deviceType)` to register handlers of their devices, and call `setCPU(CPU-arch-name)` to specify the CPU architecture. If their devices provide GPU support, developers can also inform Crystal of the GPU specification by calling `setGPU(GPU-cuda-version)`. Developers can add most commodity off-the-shelf computation devices using these simple APIs.

List 4.2 demonstrates how to add devices to Crystal.

```
// GPU server with Cuda 10.0
Crystal.registerDeviceType("2080Ti-Cuda10.0");
Crystal.setCPU("X86_64");
Crystal.setGPU("Cuda10.0");
// CPU server with X86_64 CPU
Crystal.registerDeviceType("Corei7");
Crystal.setCPU("X86_64");
// Raspberry pi 4 with aarch64 CPU
Crystal.registerDeviceType("RaspberryPi4");
Crystal.setCPU("aarch64");
```

Listing 4.2: Registering new devices in Crystal

4.4.2 Adding new hardware

Unsupported CPUs or GPUs. For devices with an unsupported type of CPUs or GPUs, such as a 32-bit CPU or a non-NVIDIA GPU, framework developers need to build customized containers for each components from scratch. Each container's specification is stored in a separate Dockerfile. A dockerfile should be named "DockerName.CPUType.GPUType."

For example, one framework developer may want to run PyTorch 1.20 on a 32-bit CPU machine without GPU support. The Dockerfile will be named "Pytorch120.X86_32.None". After that, developers only need to put these files in the default component dockerfile path in Crystal. Crystal will automatically parse the CPU type and GPU type.

Other specialized hardware. For devices with specialized hardware, for example, a specific type of FPGA-based accelerator, framework developers need to build customized containers for each processing component. A container is defined in a dockerfile that contains all execution dependencies. It is simple to Integrate these customized containers to Crystal. Developers should add their device type as an additional file descriptor right after their Dockerfiles, such as "Pytorch120.FPGA," and add these files to the Crystal component path. Developers should simply call `registerDeviceType("FPGA")` to register their devices, and call an additional `setUnknown()` to indicate to Crystal that they need to directly search for the device type when deploying an application on it.

4.5 Generating application flow graphs

In this section, we discuss how to generate flow graphs in the application assembly layer, including application flow graph construction and alternative flow graph generation.

4.5.1 Flow graph construction

The application assembly layer generates an application's flow graph as a directed acyclic graph (DAG). The basic unit of an application's flow graph is referred to as a processing component, while each processing component runs in a single container when deployed on a specific device.

Constructing a flow graph essentially means streamlining all components within this flow graph together, linking upstream components with downstream components in the DAG. As multimodal applications tend to have components that are written in different programming languages, Crystal thus provides a language-agnostic inter-component communication interface. Built upon this interface, the flow graph will be in the format of a edge set where each edge specifies an communication interface between two components.

Such an interface requires two things. First, as different programming languages has different built-in data types, this interface needs to provide a unified data abstraction to help translate all these different data types. Second, the amount of time to pass data through this interface should be lightweight, to ensure negligible overhead to the overall processing time.

Unified data abstraction. Crystal uses Protobuf [159] as the unified data abstraction format to serialize/deserialize data types in different programming languages. We use Protobuf because it is a language-agnostic, platform-agnostic tool to represent structural data. In the meantime, when comparing to serialization method like JSON [160] and XML [161], the data in Protobuf are stored in binary format, which will largely reduce the space overhead.

Lightweight data passing. Crystal establishes a shared memory circular buffer between any two components that need to communicate. Each buffer has its unique name. Any component can access this buffer by having this name and access mode (create, read-only, read-write) specified. Communicating via shared memory can minimize communication overhead by bypassing copying data to/from the OS kernel. While one processing component reads from this buffer, another can write messages to different locations, synchronized by a per-message read-write lock mechanism.

By default, each circular buffer has a length of 20 messages with dynamic message size. We set the maximum message size to 100,000 bytes to avoid unbounded sized messages.

4.5.2 Alternative flow graphs generation

As discussed in Sec. 4.2, there is implicit correlation between input data across different modalities. This correlation can be used to either improve inference quality or reduce inference latency. In particular, generating the pruned flow graphs provides additional dimensions for the application shaper layer. Crystal generates these alternative flow graphs by skipping the processing components of one or more input modalities in the original flow graph. We refer to this process as modality-based flow graph pruning. Generating alternative flow graphs makes it possible to run an application on more heterogeneous devices.

Modality-based flow graph pruning. Given the original processing flow graph, Crystal will first tag all the processing components within the pipeline using the modalities that it takes as inputs. Consider the application in Figure 4.1 as an example. The audio featurizer, face featurizer and DNN components will be tagged as "Audio," "Face," and "Audio + Face" respectively.

Having all components tagged, Crystal then prunes one or more modalties from the original data flow. If one component only takes the pruned modalities as inputs, it will be pruned accordingly. As the remaining components may depend on the output of some pruned components, the pruner sets these inputs as zero vectors with corresponding dimensions. If we revisit the example in Figure 4.1, in this case, Crystal will generate two pruned flow graphs – one with the audio modality pruned, the other with the face modality pruned.

4.6 Application shaper layer

To adapt to the hardware heterogeneity on edge, Crystal needs a large optimization space and a way to navigate this massive space efficiently. Sec. 4.5 addressed the first need by generating alternative flow graphs in addition to the original one. In this section, we focus on the navigating strategy, including a two-phase search for the configuration combination and, for DNN based applications, further model compression based on the resource setting.

More specifically, when application developers implement an application, Crystal first runs it through the offline profiling phase at the development time, where Crystal will select a subset of the whole configuration space so that the massive search space is narrowed down. Given the hardware heterogeneity, running this offline profiling on less powerful devices like raspberry pi can take unavoidably long period of time. Hence, we introduce an approximate mechanism in Crystal. In the offline profiling phase, an application only needs to be profiled once. When in the online phase, at the deployment time, we run a simple heuristic to figure out the right configuration based on the device it is deployed on.

Offline profiling. A multimodal application has several adjustable parameters, and difference combinations of parameters can influence its performance. Considering the MFN application in Sec. 4.2, adjustable parameters in MFN include language featurizer types,

video resolution, video frame rate, audio frequency, and FFT window length of the audio feaurizer. Figure 4.3a shows the accuracy-processing-time trade-off by choosing different combinations of parameters.

It is time-consuming to search through the whole configuration space each time when deploying an application. Therefore, Crystal narrows the massive configuration search space in the offline profiling phase by finding its Pareto-frontier, which is a small subset of the overall search space. For each configuration in the Pareto-frontier, one can not improve the target performance at any single dimension without worsening the performance at other dimensions. For instance, for a configuration in the Pareto-frontier of MFN, reducing the processing time always comes at the cost of accuracy reduction. The red dash line in Figure 4.3a illustrates the Pareto-frontier given the affection recognition's accuracy as the performance goal. A single data frame in MFN is a combination of a sentence in text together with corresponding video and audio frames.

Getting an accurate Pareto-frontier for an application on each individual device is not yet an efficient solution, especially for those less powerful devices, such as embedded devices. With that in mind, Crystal introduces a concept of approximate Pareto-frontier. Instead of running offline profiling for each device, Crystal runs profiling on the most powerful available device and get an accurate Pareto-frontier for that device. Based on that, Crystal includes all configurations that use the same processing time to achieve less than K% accuracy loss compared with the configurations on the Pareto-frontier to approximate the frontier. K is defined as the width of the Pareto frontier. The original results of the Pareto-frontier is a discrete configuration set. We impute the configurations between each pair of adjacent configurations using linear regression. This newly generated frontier is referred to as an approximate Pareto-frontier. Although different devices' accurate Pareto-frontier for the same application may include different configurations, an approximate frontier can actually include most of these configurations when K is set properly. Empirically, we set K as 5. Furthermore, during this profiling process, we cache the intermediate results for each processing component within a flow graph to further reduce the profiling time.

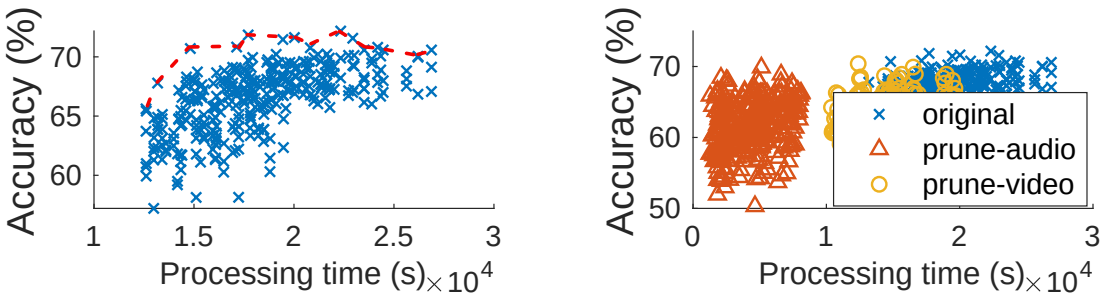
Online profiling. Having a small subset of configurations, i.e. the approximate Pareto-frontier, as inputs, in the online profiling phase, for each application on any specific device,

Crystal runs an early-terminate configuration selection algorithm to reach the performance goal under the specified constraints. For example, the performance goal and constraints can be: minimizing the processing time given less than 3% accuracy loss compared to the most accurate result. Crystal first eliminates all configurations in the frontier whose accuracy losses are larger than 3%. Within all remaining configurations, from the configuration that achieves the least to the best accuracy, we continue to run each configuration on-device until stop observing performance gain by profiling the last M configurations. Empirically, we set M as 4.

The impact of introducing alternative flow graphs. Figure 4.3b illustrates the impact of generating additional alternative flow graphs in the upper application assembly layer. For the MFN example mentioned previously in Sec. 4.2, apart from the original accuracy-processing-time trade-off shown in Figure 4.3a, there are two other sets of configurations as a result of two additional pruned data flow graphs. We can see that pruning video or audio modality away can actually provide more options for our optimizer to degrade the MFN application. For each individual flow graph, Crystal goes through the same two-phase optimization process and makes a degradation decision by jointly considering the configuration results from all flow graphs.

DNN model compression. For deep learning based multimodal applications, reduced number of modalities typically signals an opportunity for model compression and getting more performance gain. For instance, the original deep learning model in MFN takes video, audio and text inputs to perform actual classification task. In a pruned flow graph, one or more of these inputs will be set to zero vectors. It is possible that some of the parameters of the model will be multiplied by these zero-weights. Pruning these parameters from the model will not affect the inference results.

Based on this observation, Crystal introduces a pruned-modality-based DNN model compressor that hooks to an existing model compressor. First, similar to tagging in the flow graph pruner, the model compressor performs a model graph analyzing that tags each layer in a DNN model with the modalities taken as inputs. These tags are used as references for model compression in the pruning process. For example, when the audio modality is pruned, all the layers that take inputs only from the audio modality will be pruned.



(a) 324 configurations

(b) 756 configs, 3 flow graphs

Figure 4.3: Performance trade-off on 686 data frames in MFN

Crystal then compresses the target model based on the tags. Crystal hooks to a fine-grained model compressor, the Automated Gradual Pruner (AGP) [162], to prune a DNN model. There are several reasons to choose this pruning algorithm. First, it supports a per-layer pruning. During the pruning process, each layer will be pruned gradually based on the "sparsity" goal set by the pruner. The sparsity is defined as the ratio of non-zero parameters in the pruned layer to all the parameters in the original layer. With the results we got from the model graph analyzer, we can set the sparsity goals for each individual layer. Currently, we use a simple heuristic to set the sparsity for each layer. We assume that each input modality has equal contribution to the number of non-zero parameters. For instance, if one layer is tagged with text and video modalities, when text is pruned from the flow graph, this layer's sparsity goal will be set to 50%.

Second, this pruner has a few hyper-parameters and thus easy to use. There is only one per-layer sparsity parameter that needs to be set. Application developers only need to provide their model specifications and training datasets. This pruner can then automatically prune the model[2].

The existing model pruner we use has limitations in a sense that it doesn't support RNN-related compression. We can potentially choose new model pruners to mitigate this problem in the future.

2. We only presents one model pruner used in Crystal here. Many more pruners can be introduced to Crystal. The choice of pruners is orthogonal to the overall system design.

4.7 Implementation

We implement Crystal in Java and C++. Most of the application assembly layer and shaper layer are written in Java. The language-agnostic inter-component communication interface is based on the C++ boost library [163] to provide shared memory management and inter-process synchronization. We also implement multi-language wrappers for this interface that are built upon Java JNI [164], Python pybind11 [165] and Matlab MEX [166]. It takes around 2000 lines of Java code and 800 lines of C++ code to write Crystal, and all inter-component interface wrappers together total around 500 lines of code written in Java, Python, or Matlab.

Interfacing with external toolkits. For automatic model compression, Crystal interfaces with the Microsoft Neural Network Intelligence toolkit [167]. Crystal uses Docker to provide containerization support and Kubernetes to provide container orchestration support. Both Docker and Kubernetes run as separate daemons, listening for requests from Crystal.

Built-in featurizers. To maximize code reuse and minimize the domain-knowledge and development effort needed to build a multimodal application, Crystal includes a set of built-in featurizers for three commonly-seen modalities: video, audio and text.

For video data, Crystal uses FFmpeg to adjust the frame resolution and frame rate. We implement several commonly-seen image/video featurizers using various different programming languages, including Keras applications (Java-based, contain 26 pre-trained image featurizers), pretrained-models.pytorch (Python-based, contain 45 pretrained image featurizers) and OpenFace (C++-based, a facial analysis toolkit to extract more than 300 facial features) [140].

For audio, we implement widely-used featurizers, including Covarep (Matlab-based, extracting 74 different audio features from raw audio data) [139], librosa (Python-based, a package for music and audio analysis) [168] and Essentia (C++-based, an open-source C++ library for audio analysis) [169].

For text, we implement components including word2vec (C-based, an efficient implementation of the continuous bag-of-words and skip-gram architectures for computing vector representations of words) [170], GloVe (Python-based, Global Vectors for Word Represen-

tation) [171] and Google-bert-pretrained text featurizers [172] (Python-based, the state-of-the-art DNN text featurizer).

4.7.1 Extending Crystal

The layered structure of Crystal makes it easy to extend system components within one layer without having to consider other layers.

Adding new components. Crystal includes an interface to introduce a new Crystal component, making it possible for developers, either multimodal application developers themselves or third-party developers to contribute their customized processing components to Crystal. A new component can be straightforwardly defined. One simply needs to define the inputs and outputs of a component by calling `Stream()` API provided by Crystal. Reading inputs and writing outputs are straightforward by simply calling `Stream. readFrom()` and `Stream.append()`. And developers implement their customized logic in between.

Extending application shaper layer. One can add different profiling mechanisms and/or optimization algorithms to the application shaper layer by changing the profiling function in Crystal For instance, energy consumption is another important type of performance metric, and it might be the main concern for some embedded devices with limited battery life. One can integrate an external energy monitor in the application shaper layer and use the output of this monitor to enable energy-aware application shaping. This monitor can run as an extra component in addition to the components within an application. One can simply specify the energy monitoring results by using the same `setMetric()` API mentioned in Sec. 4.3.

Extending the container layer. This design makes it possible to substitute any of these two existing toolkits to other toolkits that supports additional functionalities, such as multi-tenancy that we discussed before. One only needs to change the requests' format that the newer toolkits support. For example, to switch our container orchestration framework from Kubernetes to Docker Swarm [173], one needs to change the request from Kubernetes' container cluster specification to Docker Swarm's container cluster specification.

4.8 Evaluation

Benchmark applications. We built three benchmark applications on top of Crystal: an affection recognition application (MFN) that takes video, audio and text as inputs [174], a gesture generation application (Gesticulator) that takes text and audio as inputs [175], and a multimodal neural machine translation (MNMT) application that takes image and text as inputs [176]. Each represents a unique use case of multimodal data processing.

Datasets. We use the CMU-MOSI dataset for the affection recognition application [177], the Gesticulator dataset for the gesture generation application [175], and the Multi30K dataset for the neural machine translation application [178].

Hardware. We run each application on three different devices, a GPU server with a NVIDIA 2080Ti GPU on board, a CPU server with a quad-core Intel Core i7 6700 3.4 GHz CPU with 8 GB memory, and a Raspberry Pi 4 with 8 GB memory. These devices represent typical processing units across edge and cloud, covering a wide range of heterogeneity. For large-scale experiments, we run these applications on Amazon EC2, using up to 10 instances of `m5.large`, each with 2 vCPUs and 8GB memory.

4.8.1 Ease of application development

In this section, we quantify how Crystal reduces application development effort on heterogeneous edge devices, including the development ease by using Crystal built-in modules, and adding customized modules.

Crystal applications are easy to implement. Table 4.1 displays the number of lines of code needed when writing three benchmark applications with and without Crystal. Instead of building everything from scratch, a developer only needs to write less than 30 lines of code to build an end-to-end application for deployment across our experimental settings. This translates to development effort reduction by a factor of 5 to 10.

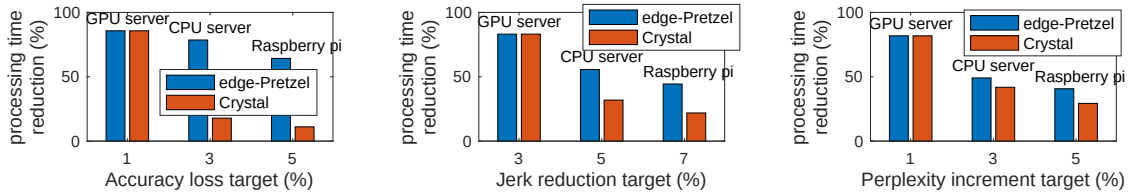
Crystal framework is easy to extend. Sec 4.7 explained extending Crystal in different ways. Here, we highlight the ease of adding new featurizers to Crystal. We anticipate this to be the most common extension to Crystal. We count the number of lines of code to build each built-in featurizer in Crystal. Across all the built-in featurizers mentioned in Sec 4.7,

Table 4.1: Lines of code needed to implement applications.

Application	Flow graph		DNN
	w/o Crystal	with Crystal	
MFN	3346	25	542
Gesticulator	3299	21	617
MNMT	2260	20	220

it requires less than 30 lines of code on average to add to Crystalbased on an existing processing function, which consists of a proto file used to specify input/output message, a yaml file that contains all configurable parameters, the code to specify the input/output streams and the specific feature extraction logic.

4.8.2 Graceful performance degradation



(a) MFN (b) Gesticulator (c) Multimodal NMT

Figure 4.4: Minimizing processing time given accuracy constraints on various devices

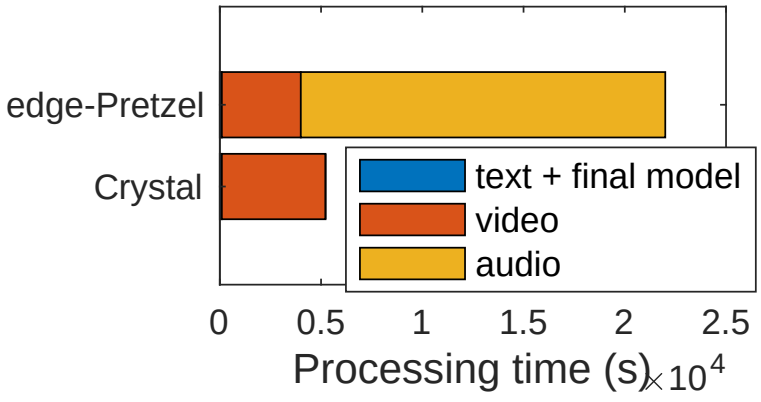
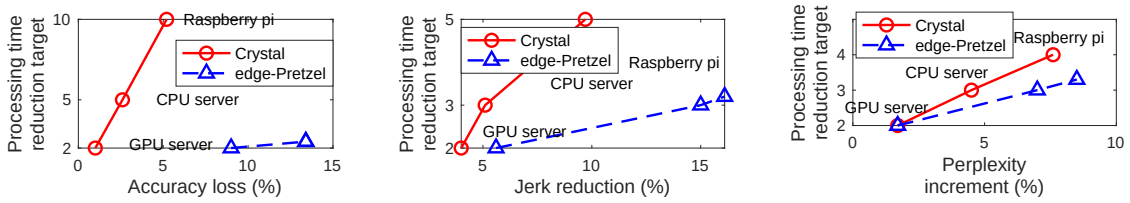


Figure 4.5: Processing time breakdown of the CPU-server performance in Figure 4.4a

In this section, we evaluate the performance of three different applications running on heterogeneous devices, showing Crystal's ability to support automatic degradation. For comparison, we also implement cross-component performance optimization in Pretzel [152],



(a) MFN (b) Gesticulator (c) Multimodal NMT

Figure 4.6: Maximizing accuracy given processing time constraints on various devices

which essentially conducts performance optimization without alternative flow graphs. We refer to this as edge-Pretzel. In addition to running Crystal on a single device, we run Crystal with Kubernetes to scale up to more than one Amazon EC2 server. Crystal supports a continuum, from cloud to a wimpy embedded device, either degrading gracefully based on the hardware capability or scaling up seamlessly through interfacing with Kubernetes.

Single-modal applications. We will start from the simplest case – a single-modal application. We build a PyTorch-based VGG19_BN [145] object recognition application atop edge-Pretzel and Crystal respectively. Then we deploy it on the GPU server, Core i7 CPU server and Raspberry pi 4.

As Pretzel was originally designed for the Cloud setting with only CPU support, the edge-Pretzel-based application can only run successfully on the Core i7 CPU server. When setting up the same performance goal of "minimizing the processing time with less than 3% accuracy loss," Crystal-based and edge-Pretzel-based applications achieve the same performance (around 160 seconds to process 1000 images) on the CPU server. With the help of containerization, the Crystal-based application can also run on the GPU server and Raspberry Pi 4 respectively.

For single-modal applications, the same notion of degradation works for both Crystal and edge-Pretzel when running on homogeneous devices. But this task can not be fulfilled without proper system-level support for heterogeneous devices. In the following experiments, we couple edge-Pretzel with the basic containerization support provided by Crystal, making it possible to run on heterogeneous devices.

Minimizing inference time. Moving to multimodal analytics, we run three multi-modal applications on heterogeneous devices and set the accuracy constraints accordingly for dif-

ferent devices. The overall intuition is to relax the accuracy targets on less powerful devices. Figure 4.4 shows the results. Different applications incur different levels of accuracy loss. For the accuracy metrics shown in all three X-axes, a larger value on the X axis represents a higher level of accuracy loss. There are three sets of bars in each subfigure, from left to right, representing the application performance running on the GPU server, the CPU server and the Raspberry Pi respectively. We use the inference time that the most accurate configuration takes on each device as the baseline and normalize the end-to-end inference times.

For all three applications, edge-Pretzel and Crystal achieve similar processing time performance when running on the GPU server, but Crystal outperforms the edge-Pretzel by up to 10× when running on the CPU server and the raspberry pi. The main reason is that generating alternative flow graphs in Crystal is expected to expand the trade-off space at the cost of reduced accuracy. Relaxing the accuracy constraints makes it possible for Crystal to find a better configuration to degrade the application.

Zooming in, when considering running MFN on the CPU server with the accuracy constraints "less than 3% accuracy loss," Figure 4.5 shows the processing time breakdown of the Crystal and edge-Pretzel versions of MFN. We can clearly see that Crystal outperforms edge-Pretzel by wisely pruning all the audio-related processing. Moreover, when considering video processing in both cases, Crystal assigns even more time compared to edge-Pretzel. In other words, Crystal strategically assigns the limited resources to the processing of more important modalities.

Maximizing accuracy. It is a simple process to specify performance goals in Crystal. With only two lines of code change (one line for the goal, the other for the constraint), Crystal can switch from targeting processing time minimization to inference result accuracy maximization. Figure 4.6 shows the results of maximizing accuracy given processing time constraints. The X-axis represents the accuracy loss for each application running on different devices, while the Y-axis shows the processing time reduction target (in terms of the reduction factor). Similar to Figure 4.4, we set different processing time goals for different devices. For instance, when application developers expect to see a processing time reduction by more than a factor of 5, they will set this processing time reduction target to

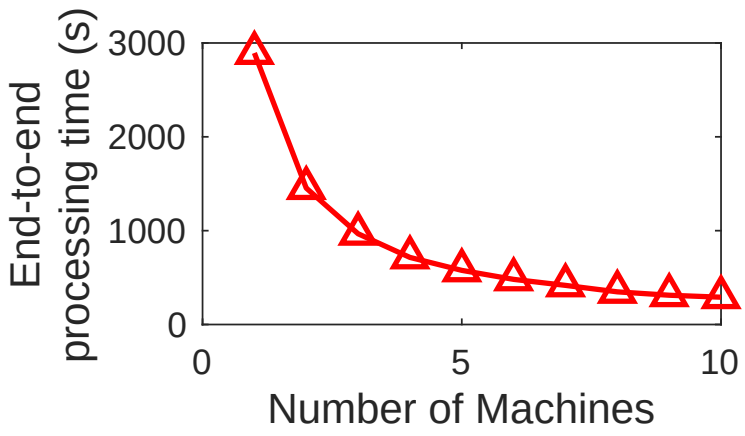
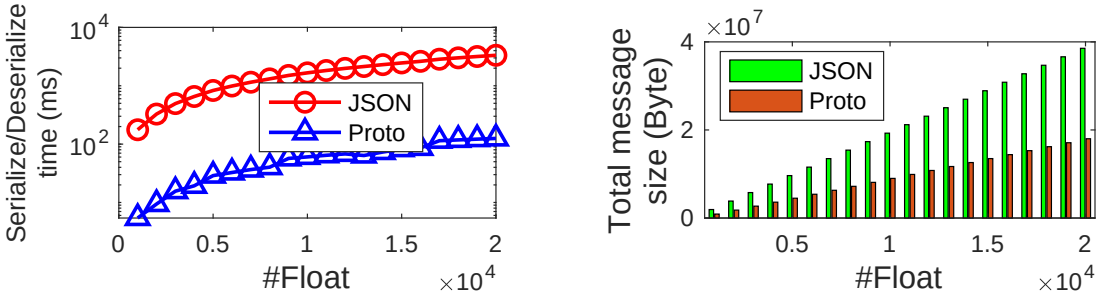


Figure 4.7: Large-scale end-to-end example: Take Gesticulator as an example



(a) Serialization/Deserialization time

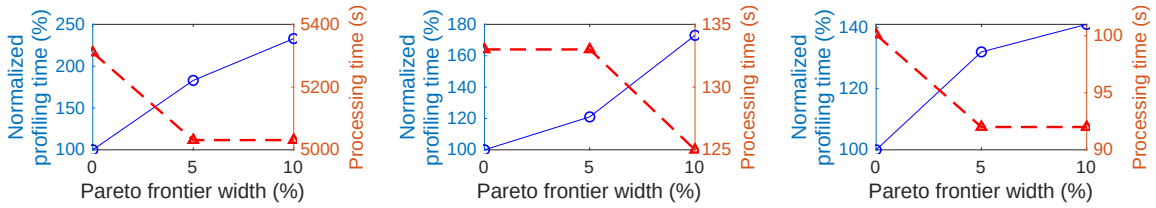
(b) Message size

Figure 4.8: JSON vs Protobuf

5.

Considering the most accurate version of each application as the baseline, Crystal manages to degrade each application to satisfy the processing time goal with less than 10% accuracy loss. Compared to edge-Pretzel, Crystal achieves the same processing time goal with less accuracy loss when the processing time reduction target is small. Moreover, when running on less powerful devices like the CPU server or raspberry pi, with limited capability to trade accuracy for processing time, edge-Pretzel fails to fulfill some of the performance goals when Crystal can manage to generate a suitable degraded application. For such cases, Figure 4.6 shows edge-Pretzel's best-effort performance.

Scaling up. Figure 4.7 shows the performance of running Gesticulator on 1 to 10 Amazon EC2 m5.large instances. As we can see from the figure, the processing time is inversely proportional to the number of EC2 machines. We also run MFN and MNMT using the same setting, and observe similar performance results. The configuration of Gesticulator



(a) MFN (b) Gesticulator (c) Multimodal NMT

Figure 4.9: Profiling time and performance difference given different Pareto frontier's width

Table 4.2: MFN model pruning

Model	#Non-zero Parameters	Sparsity (%)	Accuracy (%)
Original	7.12×10^5	100	72.3
Audio-based compression	5.82×10^5	81.06	71.3
Video-based compression	5.82×10^5	81.06	71.2
A&V-based compression	4.46×10^5	62.12	70.1

is profiled based on a single EC2 instance. All components within Gesticulator are encapsulated into a single Pod in Kubernetes. With Kubernetes, one can scale up Gesticulator from 1 to 10 machines by simply set the number of replications in the Kubernetes yaml configuration file. Each workload is first sent to the control plane of Kubernetes, and Kubernetes can automatically handle load balancing and restarting failing EC2 machines. All these processes are transparent to application developers.

4.8.3 Microbenchmarks

In this section, we evaluate three important system components to show the trade-off when designing them, including the modality-based DNN model pruner, the language-agnostic inter-component communication interface and the end-to-end performance optimizer.

Table 4.3: Gesticulator model pruning

Model	#Non-zero Parameters	Sparsity (%)	Jerk (cm/s^3)
Original	3.08×10^5	100	821
Text-based compression	1.87×10^5	60.78	750

Modality-based DNN model pruning. Tables 4.2 and 4.3 show the automatic model compression results given different pruned modalities. We omit the compression result of the MNMT application here. As the inference model used in MNMT is a recurrent neural network (RNN), which can not be further pruned by our model pruner.

As we can see from these two tables, our model pruner can prune up to 40% of the total number of nonzero parameters while incurring less than 3% accuracy loss for MFN and less 10% Jerk reduction (Jerk is a metric to quantify gesture quality, the higher the better). We omit text-based model pruning results for MFN and audio-based model pruning results for Gesticulator, as we found that pruning these modalities will affect the accuracy/jerk significantly. In the Crystal runtime, these pruned models will not be considered as viable candidates.

Language-agnostic inter-component communication. One key source of overhead from language-agnostic inter-component communication is data serialization. Figure 4.8 shows the time and space overhead differences between two commonly-use serialization techniques, JSON serialization and Protobuf serialization, the latter of which is used in Crystal. We use these techniques to serialize a variable-length vector of floating-point numbers, as floats are the representative data type passed between components. For both subfigures, the X axis is the number of floating numbers in each vector. Figure 4.8a shows the overall serialization/deserialization time, while Figure 4.8b shows the total message size after serialization. Protobuf requires two orders of magnitude less serialization time, while the messages generated requires only half the space compared to those from JSON.

Further, Table 4.4 shows the inter-component communication overhead. First, We fill a message queue with 1,000 messages. Each message contains a random number of floating numbers, ranging from 1,000 to 10,000. After generating this message queue, we let components written in different programming languages pass these messages between each other and record the elapsed time. Most of these communication times are less than 1 ms per message. Even considering the most heavy-weight communication among two components written in Matlab, it takes on average 3 ms to pass a single message.

For a baseline, we also pass the same set of messages between two Java modules or two Python modules, via the built-inJava and Python inter-process interfaces. The java-java

Table 4.4: Cross-language cross-module communication time

Time (s)	Java	Python	C++	Matlab
Java	0.75	0.71	0.72	2.95
Python	0.66	0.26	0.26	2.31
C++	0.61	0.26	0.19	2.30
Matlab	2.86	2.48	2.49	3.10

communication time is 0.15s, while the python-python communication time is 0.03s. The difference is mainly from the data serialization/deserialization process in Crystal.

End-to-end performance optimizer. Given that we are using an approximate Pareto-frontier to estimate all potential configurations when running on different devices, the width of this frontier determines the overall performance. Figure 4.9 illustrates this trade-off. When setting the performance goal as minimizing the processing time on the CPU server with less than 3% accuracy loss, with higher Pareto frontier width, all three applications can see a processing time reduction. However, this procedure comes at the price of as high as double the online profiling time. Based on this observation, by default, the Pareto-frontier width of our performance optimizer is set to be 5%, aiming to balance between profiling time and application performance.

4.9 Related Work

System support for smart spaces. Both academia [141,179,180] and industry [181–183] recognize the need for system support for smart spaces. Previous works address the device heterogeneity challenge by providing unified device abstractions [184, 185] and common policy languages to ease application deployment across heterogeneous devices [183]. These take the first step of addressing deployment challenges, but do not deal with heterogeneous *computation power*. In contrast, Crystal provides not only a unified deployment interface, but automatically adapts to myriad processing capabilities transparent to the application developers.

Multimodal learning. Multimodal learning, or the earlier *multi-sensory fusion*, extracting insights from more than one modality of input data, supports wide-ranging use cases such as affective computing [186], speech recognition [138] and activity recognition [137], of

interest to both academia [136, 187–189] and industry [190]. These works mainly focus on deriving suitable inference models. A recent work [191] aims to address the imbalance in the ingestion rates among different modalities. In comparison, Crystal provides generic system support to simplify application development and deployment in diverse settings, without modifying the semantics of the inference logic.

Stream processing engines. Streaming data analytics have been explored in both cloud and edge settings [192]. DS2 [193], intended for the cloud, automatically estimates the true processing and output rates for each operator within a streaming dataflow to facilitate dynamic resource allocation. EdgeWise [194] is designed for multiple streaming dataflows sharing the same edge devices; it employs a congestion-aware scheduler to achieve high processing throughput. Both systems focus on data processing efficiency, but Crystal focuses more on mapping that to diverse hardware capability.

Inference serving frameworks. Several machine learning and deep learning serving frameworks have been developed in recent years for model deployment [148, 152, 195–197].

Clipper [148] treats a single machine learning model as a black-box and only applies pipeline-agnostic optimizations such as caching and batching, offering little opportunity to deal with hardware heterogeneity. Inferline [196] then extends support to a multi-model inference pipeline. Pretzel [152] aims to gracefully degrade the inference performance as multiple concurrent inference pipelines compete for resource in the cloud. Nexus [195] targets inference serving on a GPU cluster. All these works abstract away the analytics applications (single-modal or multi-modal) into the inference models, while Crystal provides more system support within a multimodal application. Further, these focus on the cloud, while edge scenarios may involve resource capability that differs by orders of magnitude and the available mechanisms for the cloud cannot deal with that level of disparity.

Mensa [197] is a recent edge-centric system for serving a single DNN model on various mobile accelerators. However, it does not handle other end-to-end application deployment support as provided by Crystal.

4.10 Summary

While edge analytics scenarios have grown in complexity due to increasingly sophisticated multimodal inference logic and heterogeneous hardware execution settings, there is little generic system support to ease application development.

Therefore, we design and implement Crystal in this chapter. Crystal refactors the application processing flow into three abstraction layers, masking hardware heterogeneity with abstract resource types through containerization, and abstracting complex application logic into dataflow graphs. In doing so, Crystal then *shapes* the application flow based on the resource availability and degrades gracefully and automatically. Crystal is shown to reduce application development effort substantially, and we believe it can facilitate emerging edge application deployment.

Chapter 5

Video-zilla: An Indexing Layer for Large-Scale Video Analytics

5.1 Introduction

Recent years have witnessed a sharp uptick of the number of video cameras. 2018 alone saw 140 million shipments of video cameras [33]. These are widely deployed for monitoring and surveillance in diverse sectors, ranging from traffic control [198], crime investigation [199], to healthcare [200]. For example, the city of Bellevue, WA has deployed cameras at many intersections to monitor traffic; According to the British Security Industry Authority [34], approximately 300,000 cameras are already deployed in UK schools. These surveillance applications have generated a wide variety of video analytics workloads. At the core of these workloads is a set of computer vision tasks, such as object detection and tracking, which then form the basis for specific applications such as license plate identification, tracking suspects, and fall detection for elder care.

While video content analysis dates back decades, today's large-scale video analytics landscape is drastically different. First, video analytics used to be run on a small number of static video clips (i.e., pre-recorded with a fixed number of frames); Instead, we now have many surveillance cameras generating continuous video feeds. A single camera alone capturing video at 30 frames per second can generate 20 GB of video per day [201]. Sec-

ond, the computer vision tasks have also grown in sophistication thanks to the increasing adoption of deep learning inference. Applying a state-of-the-art object detection DNN in real time (i.e., processing 30+ frames per second) to a single video feed requires a powerful GPU [202] costing $4000. Further, it is increasingly common to aggregate and analyze a large number of video feeds [203], for example, to track a suspect car around a city. The sheer volume of the videos today presents colossal scalability challenges to large-scale video analytics systems.

Existing large-scale video analytics systems tackle the daunting scalability prospects by leveraging the inherent redundancy within video feeds [202, 204, 205]. For example, objects in successive frames are likely to be the same, exhibiting *frame-level redundancy*; Cameras deployed at the same intersection often capture the same vehicles and pedestrians, though from different perspectives, due to the *spatial-temporal correlation* of camera placement and captured scenes. The former optimizes for individual frame retrieval, oblivious to the collective semantics of a feed, while the latter optimizes across video feeds, though requiring manual labeling. Redundancy elimination combines naturally with edge processing [36] and large-scale video analytics is a killer app for edge computing [203].

Still, neither redundancy management strategy can keep pace with the growing number of video feeds. Inter-feed analytics jobs effectively need to profile each camera feed individually. Fundamentally, a camera feed comprises of *scenes*, each carrying an implicit, collective sense of content semantics, such as *parking lot*, *downtown*, and *school*. This semantics dictates the type of objects and events featured in the feed. Exposing this semantics facilitates further optimizations to process the most relevant (subsets of) feeds. However, there are few mechanisms available to recognize this collective semantics and organize feeds. Feeds today are mainly stored in a file system following common encoding formats. Recent video analytic systems each implements application-specific frame profiling and sampling strategies to reduce redundant data processing. (Section 5.2)

In this chapter, we build *Video-zilla*, a standalone indexing layer interposed between video query systems and a video store to organize large-scale, multi-camera feeds. We propose a video data unit abstraction called *semantic video stream (SVS)* based on a notion of distance between objects in the video (Section 5.3). SVS implicitly characterizes the video

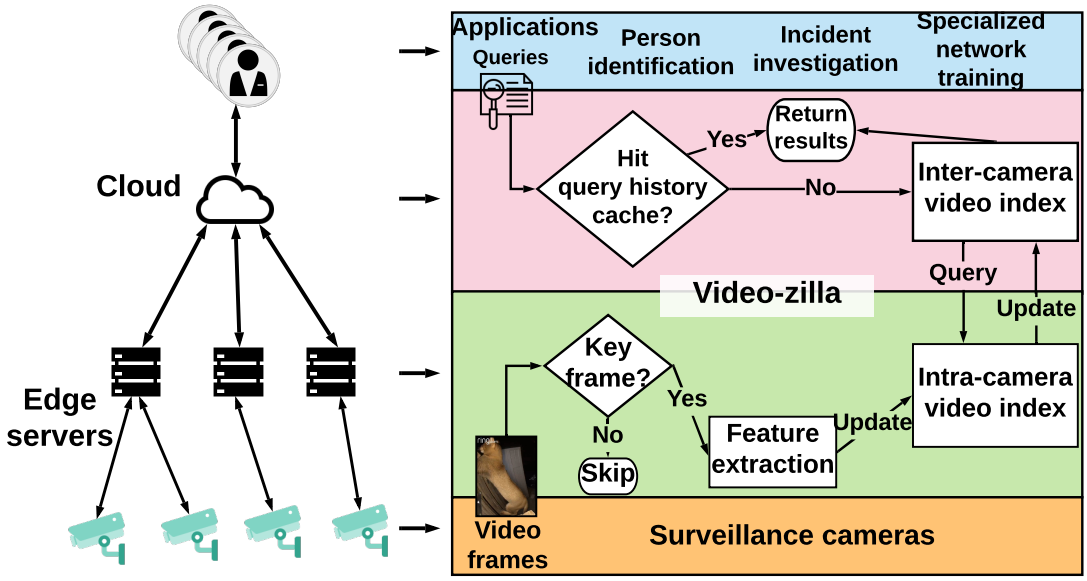


Figure 5.1: *Video-zilla* architecture

content by *scenes*, which is missing from current video data abstractions and offers a suitable middle-ground granularity between individual frames and an entire camera feed. We then build a hierarchical index that analyzes and exposes the semantic similarity both within and across camera feeds (Section 5.4). This way, *Video-zilla* can quickly organize video feeds based on their content semantics without manual analysis of video content, while preserving the boundaries between cameras (Section 5.5). *Video-zilla* provides a generic service to a range of query systems. The index can be run either offline over static videos or online over live feeds with new frames arriving continuously. The built-in automatic video segmentation naturally delineates different scenes, which is also useful for moving cameras like dashcams or drones capturing frequent changing scenes. Finally, we refactor existing large-scale video analytics pipelines to separate out frame content analysis as an independent module, providing a per-camera video ingestion interface to the underlying surveillance cameras and a centralized query interface to upper-level video query engines (Section 5.6).

Video-zilla effectively acts as an *explanation database* [206–209] for the video data, and a similar approach can apply to other unstructured but correlated IoT data. We evaluate *Video-zilla* with three case studies, object identification, specialized network training and video archiving (Section 5.7). Compared to existing indexing strategies, *Video-zilla* reduces

query resource usage by up to 14× and can offer video clustering or archival support that is currently not feasible. *Video-zilla* can replace substantial code development for video content analysis in existing analytics systems with a few lines of query code.

In summary, we make three contributions: First, we propose Semantic Video Stream (SVS), a novel video data unit abstraction based on a notion of distance between objects in the video, which abstracts away the video content precisely and concisely. Second, based on SVS, we build a hierarchical video index as a data organization strategy to expose the correlation between and within camera feeds. By preserving the boundary between cameras while capturing their correlation, this lends to easy incorporation of additional cameras or policies such as privacy measures. Third, we advocate for refactoring current video analytic systems into a generic indexing layer, *Video-zilla*, and the specific analytics queries on top of that. *Video-zilla* can simplify building analytic applications and reduce time complexity of inter-camera video analytics to sublinear with the number of camera feeds.

5.2 Tackling the scalability challenge

Consider a canonical multi-camera deployment (Figure 5.1). For surveillance applications, the objects in the videos constitute the main content of interest, hence the "video semantics". Video analytics systems sift through the video feeds for certain objects. Given the high data volume and the low ratio of *signal* (useful objects) to *noise* (useless background scenes), efficient analytics rests on pruning the search space fast. Existing systems are still far from scaling analytics sublinearly with the number of feeds and frames.

5.2.1 Lack of adequate video data abstraction

Video data are gigantic but highly redundant. The key enabler of a scalable multi-camera video analytics system is to fully understand and leverage this inherent redundancy. However, there is insufficient understanding and characterization of the redundancy in the current multi-camera video surveillance landscape, impeding further optimizations.

Types of redundancy within videos. Figure 5.2 shows 4 consecutive images captured by the same camera. We can always see the same car bounded by the red box across

all these frames. This is the type of *frame-level similarity* already leveraged extensively in video coding and analysis [202, 204]. Figure 5.3 shows snapshots from two live video feeds captured by surveillance cameras mounted at the same intersection in Jackson Hole, WY [210]. We can see significant overlap between the views of these two cameras. This is a form of *camera-level spatial-temporal similarity* [203, 205] due to the physical camera placement. Figure 5.4 shows two images captured by video cameras installed in different parking lots from the VIRAT dataset [211]. Unlike that in Figure 5.3, spatially these two cameras are not necessarily near each other. However, since both can be viewed as "parking lot" cameras, the video feeds captured by these two cameras are somewhat similar to each other. For example, both of them might contain cars and people stepping out of their cars. We refer to this as *stream-level semantic similarity*, where a *stream* is a contiguous block of frames within a camera feed. We make a distinction between "camera-level" and "stream-level" since a long camera feed may be divided into multiple streams, each featuring different objects (hence "semantics"). As a simple example, consider the video feed from a camera mounted near a railway track. The frames capturing an approaching train are semantically different from when no train is present.

Existing views and limitations. Current data organization in video query systems can be seen as an extension of image retrieval systems, built on top of the *frame-level similarity*. All video data are treated as a giant collection of video frames. However, this requires processing every frame within the video collection at ingestion or query time, which incurs incredibly high computation overhead. While frame sampling or adding camera-level constraints [36, 204, 205] can reduce the number of frames processed, the number of frames extracted from the streaming video data is still huge.

A new abstraction. Clearly, there is a mismatch between existing data abstractions and the fundamental video data characteristics. This necessitates a mechanism to aggregate video at a sub-feed level to both accurately preserve the frame-level video content and minimize the number of data objects to handle. We therefore argue for a level of abstraction between the feed level and the frame level.

We propose a new data abstraction, semantic video stream (SVS), that can represent the content of a subset of a feed. Video frames from a single camera can be divided into several

Figure 5.2: Images captured by the same camera

Figure 5.3: Images captured at the same intersection

video streams based on their content difference, likely due to scene changes. SVSs aggregate frames to drastically prune the search space. Besides, SVS-based data organization lends to large-scale inter-feed processing that is currently difficult. Examples are specialized neural network training and aggressive video archiving. A semantic understanding of video streams makes it possible to train one specialized neural network per-cluster of SVSs. For the latter, instead of observing the access pattern of individual frames, we can operate at the level of an SVS or a cluster of them.

5.2.2 Multi-faceted policy considerations

The rise of edge data source. As data collection is increasingly at the edge of the network [75], the data volume at the edge is growing exponentially. Further, given how

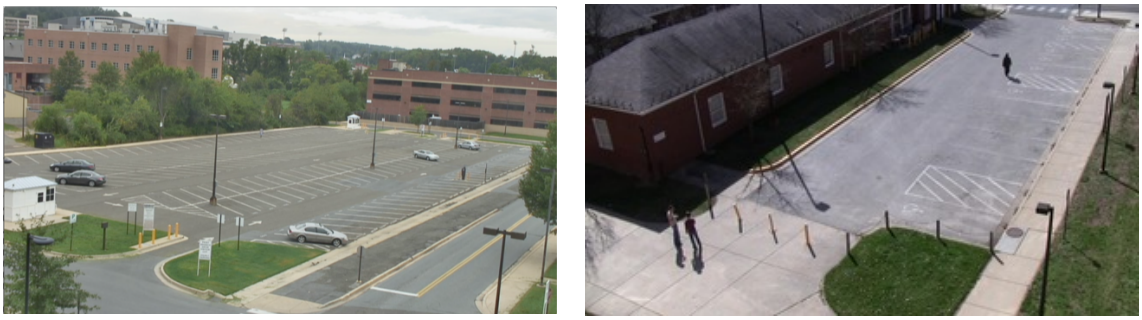


Figure 5.4: Images captured in different parking lots

pervasive surveillance cameras are deployed, the captured data tends to raise privacy concerns [212]. Feeds from different cameras may need to be isolated depending on who owns the camera and manages the captured videos. It is desirable to minimize the number of raw frames sent both for bandwidth and privacy considerations.

Our approach. We organize video data using a hierarchical index, with inter-camera and intra-camera components. The hierarchy can be naturally mapped to different processing locations. Instead of sending all raw data to the cloud, we can send only the representative video streams, and only raw frames containing objects of interest can leave the edge processing point near a camera. Besides largely reducing network overhead in the backbone network, this offers privacy support by preserving the data boundary and a summary view across cameras to the cloud to speed up subsequent query processing.

5.2.3 Intertwined management and analytics

Existing video stores treat video data the same as any data, and do not provide any support to characterize the redundancy. This shifts the burden to the query systems. Recent video analytics systems implement built-in custom strategies to reduce redundant data processing and scale analytics [36, 204, 205].

Scaling implementation efforts. Building an effective video index requires notable domain knowledge of video processing [203]. Apart from the merit of the processing strategies, including an index within each system inevitably requires repeated implementation of common processing steps, which itself is inefficient. Further, this indicates a lack of separation between basic data organization and actual analytics that makes it harder to scale in other ways.

Towards a data organization layer. Instead, video data analysis and organization should ideally be a separate layer overlaid on a video store and support common types of video queries. For video surveillance, two common types revolve around *specific object identification* (*direct queries*) and *correlating camera feeds* (*clustering queries*). The former takes the format of *find video streams that contain object X in N streams.* This forms the basis for various applications like *theft detection* [213], *incident investigation* [214] and even *atmospheric administration* [215]. The latter is in the form of *find all video streams that*

are semantically similar to a video stream Y, which lends to effective specialized network training and video archiving. Additional qualifiers over a subset of camera or time range can be easily supported.

5.2.4 Introducing *Video-zilla*

So far, we have motivated the need for a new abstraction, a hierarchical index exposing boundaries between cameras, and a system to separate and interface with the video store and the query engine. *Video-zilla* is designed to address these. We outline the challenges and solutions below.

Deriving semantic video streams. Previously, we gave a qualitative illustration of two semantically similar video streams. To be integrated into an analytic system, we need a suitable metric, i.e., a quantifiable definition of "semantics", to delineate SVSs to provide an effective abstraction. On that basis, we need to measure the "similarity" between SVSs for further aggregation. Section 5.3 describes representing a semantic video stream as a feature map and computing the similarity score between two streams based on a novel metric, *object mover's distance*. We also introduce an approximation method to reduce the computation overhead.

SVS clustering for index construction. We need an index that is precise and concise, capturing the video content comprehensively as well as the level of redundancy. This requires low-latency and effective clustering of all SVSs incrementally as new SVSs arrive in a streaming fashion (Section 5.4).

Expressive interfacing. As a generic indexing layer between a video store and query applications, *Video-zilla* should provide clean and expressive interfaces to hide the complexity of the underlying data organization but capture the video content. Sections 5.5 and 5.6 describe the architecture of *Video-zilla*, the functionality it provides, and the APIs exposed.

5.3 Semantic video streams (SVSs)

In this section, we define an SVS and a quantifiable notion of "semantics" (Section 5.3.1). We then propose a novel metric, *object mover's distance (OMD)*, to compare the seman-

tics among different video streams, as well as an approximate algorithm (Section 5.3.2) to speed up OMD computation. Finally, we discuss how to represent several semantically similar SVSs using a representative SVS (Section 5.3.3), a primitive for subsequent indexing operations.

5.3.1 Defining Semantic Video Streams

We define a *video stream* as a contiguous block of frames within a camera feed. The "semantics" of a video stream describes the content of these frames. For object-identification based surveillance applications, the content of each frame can be characterized by the objects captured in that frame. Therefore, we characterize the *semantics* of a video stream with all the objects within that stream, with per-object feature vectors. A *semantic video stream (SVS)* then is the collection of these feature vectors (i.e., the *feature map*).

Note that an SVS only captures the object distribution that may correspond to an event or scene change, but cannot and does not aim to explicitly identify the event. For example, a train-station camera mainly captures two types of scenes interleaved with each other: *train passing*, and *empty tracks*. These will be mapped to different SVSs, some featuring trains. An object distribution of "85% train and 15% people" may correspond to "train passing". The analytics application using these SVSs may *infer* the corresponding events if provided with additional contextual information about the video. Further, an SVS cannot track object motion, which requires tracking more frame-level information than the object distribution. **Video frame clipping.** In preparation for deriving SVSs, we first clip objects that is contained in each frame using an object detection mechanism. The notion and derivation of SVS are orthogonal to per-frame object detection, so we can use either lightweight ones such as the YOLO series [216] or more sophisticated approaches [217–219] to balance the computation footprint and detection accuracy trade-offs. In our evaluation we use YOLO. Each object is represented by its four-point 2-D coordinate (top, left, bottom, right) in the original frame.

Object feature extraction. To characterize an SVS, the first key step is to obtain the feature vector per object. We can lean on state-of-the-art image classification tools such as convolutional neural networks (CNNs) [220–222]. The output of a CNN is the probabilities

of all object classes, and the class with the highest probability is the classification result. The penultimate layer's output of a CNN proves to be representative features of an input image [223]. The features form a real-valued vector, whose length ranges from 512 to 4096 in the latest CNNs [145,220]. Therefore, running a CNN until the penultimate layer on any image containing an object extracts the feature vector for that object. For a video frame with multiple objects, we can clip the frame and obtain the feature vectors for all objects within the frame.

5.3.2 Distance between SVSs

The key to our data indexing system is to cluster semantically similar SVSs to reduce the search space for queries. Therefore, we need a mechanism to quantify the similarity between SVSs.

SVS distance. Since the SVS semantics refers to the objects captured, "semantic similarity" between two SVSs manifests as similar objects and their distributions in different SVSs. An SVS distance metric should then reflect how far object distributions deviate between SVSs, i.e., the *travel cost* between objects in different SVSs. As objects are represented by feature vectors, one natural measure is the Euclidean distance in the feature vector space, defined as $d(i,j) = ||\boldsymbol{x}_i - \boldsymbol{x}_j||_2$, where $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are the feature vectors of objects i and j.

The "travel cost" between two objects is a natural building block for the distance between two SVSs. Let $\boldsymbol{d}$ and $\boldsymbol{d'}$ be two feature maps, containing n and n' feature vectors, respectively. We allow each object i in $\boldsymbol{d}$ to be either partially or fully transformed into any object in $\boldsymbol{d'}$. Let $T \in \mathbb{R}^{n \times n}$ be a cost matrix where $T_{ij} \geq 0$ denotes the unit travel cost from object $i \in \boldsymbol{d}$ to object $j \in \boldsymbol{d'}$. To transform $\boldsymbol{d}$ entirely into $\boldsymbol{d'}$, we need to ensure that the overall outgoing flow from object i equals d_i, i.e., $\sum_j T_{ij} = d_i$. Further, the amount of incoming flow to object j should match d'_j, i.e., $\sum_i T_{ij} = d'_j$. This way, we define the distance between two SVSs as the minimum cumulative object travel cost required to move all objects in $\boldsymbol{d}$ to $\boldsymbol{d'}$, i.e., $\sum_{i,j} T_{ij} d(i,j)$.

Object Mover's Distance. Calculating the aforementioned minimum cumulative object travel cost can be formalized as:

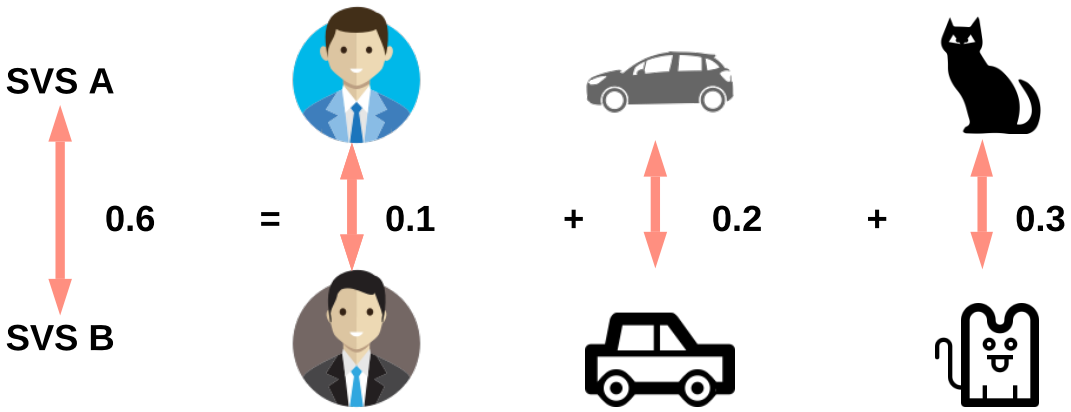


Figure 5.5: Illustration of Object mover's distance

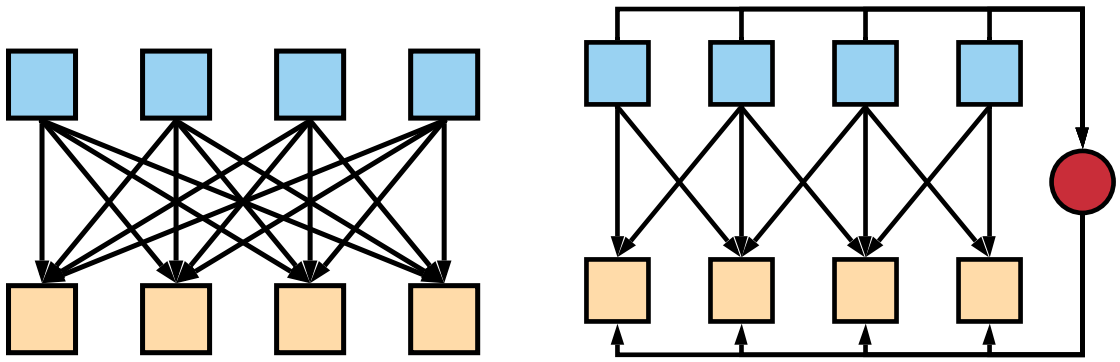
$$\min_{\boldsymbol{T}\geq 0} \sum_{i,j=1}^{n} \boldsymbol{T}_{ij} d(i,j) \text{ where } : \sum_{j=1}^{n'} \boldsymbol{T}_{ij} = \frac{1}{n}, \forall i \in 1, ..., n \\ \sum_{i=1}^{n} \boldsymbol{T}_{ij} = \frac{1}{n'}, \forall j \in 1, ..., n' \tag{5.1}$$

$d_i = 1/n$ and $d_j = 1/n'$ mean that we treat every feature vector within an SVS equally by giving them the same weight. The weights in each SVS sum up to 1. Equation 5.1 aims to minimize the weighted cost of transferring ("mapping") all feature vectors in $\boldsymbol{d}$ to $\boldsymbol{d'}$. A vector i in $\boldsymbol{d}$ could be mapped to several vectors in $\boldsymbol{d'}$, with its weight d_i split over the target vectors' in $\boldsymbol{d'}$. Therefore, the mapping is one-to-many, not one-to-one, as shown in Figure 5.6a.

This is a special case of the well-studied transportation problem, Earth Mover's Distance [224], with specialized solvers [225,226]. To highlight this connection, we use the term *object mover's distance* (OMD). OMD is a metric since $d_{i,j}$ is a metric [227]. Figure 5.5 illustrates the OMD metric for two SVSs, A and B. Each arrow represents the correspondence (or *flow*) between two objects, their Euclidean distance shown next to the arrow. The distance between two SVSs reflects a cumulative travel cost across the objects.

Fast OMD computation. Calculating an accurate OMD between two SVSs has $O(n^3 \log n)$ time complexity, where n is the number of features within a video stream. It is thus impractical to compute the OMD between large video streams.

Fortunately, there are several fast specialized approximate computation methods [225, 226, 228]. We adopt the thresholded ground distance method [226]. Instead of comparing



(a) Original OMD (b) Thresholded OMD

Figure 5.6: Illustration of the thresholded OMD

all pair-wise object moving costs, we set a distance threshold t. Any pair-wise distance between feature vectors larger than t will be capped to t. We essentially merge all such pair-wise relationships to a single group.

Figure 5.6 illustrates the transformation process from the original OMD computation to the thresholded-OMD computation. The boxes are objects and the colors indicate different video streams. The red circle is the new transshipment vertex that serves as an intermediate vertex, before a feature vector goes to its original destination. Any incoming edge cost is the threshold t and any outgoing edge cost is 0. The drastic reduction of the number of edges is the main reason for computation time reduction when using FastOMD.

5.3.3 Representative construction

To efficiently cluster SVSs, we also need a way to construct a representative SVS for an SVS cluster. A new SVS can then be compared with per-cluster representatives, rather than with all SVSs in that cluster. Section 5.4 explains how to identify a suitable existing cluster for a new SVS to join.

k-clustering representative construction. Recall that objects that are visually similar have similar feature vectors in the Euclidean space. We perform k-means to derive k clusters of feature vectors, each has a centroid feature vector. Weighted by the size of each cluster, these centroids‘ vectors then form a representative SVS.

We adopt the silhouette method to determine the value of k [229]. The silhouette value

is a measure of how similar an object is to its own cluster compared to other clusters. Formally, for data point $i \in C_i$, $s(i) = \frac{b(i)-a(i)}{max\{a(i),b(i)\}}$, where $a(i) = \frac{1}{|C_i|-1}\sum_{j \in C_i, i \neq j} d(i,j)$ and $b(i) = \min_{k \neq i} \sum_{j \in C_k} d(i,j)$. A high silhouette value indicates a well-formed cluster.

Defining query hit. To facilitate direct object identification queries, we record the boundary for each weighted center. The boundary is defined by the distances between the farthest data points in all directions and the cluster center. We obtain a query *hit* when the queried feature lies within the boundary of a weighted center.

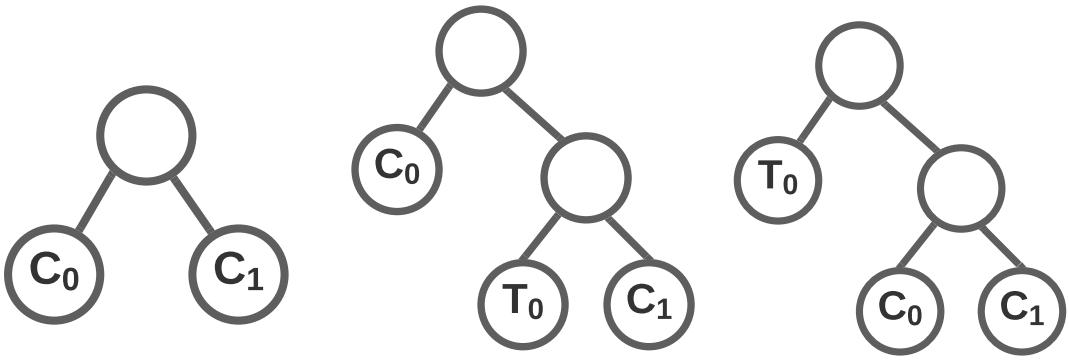
5.4 Clustering SVSs

Given a set of SVSs, index construction for fast query boils down to clustering the SVSs based on the similarity of their semantic content. In this section, we first explain an existing incremental clustering algorithm that works for both the Euclidean and OMD metric space, explaining the basic data structure and related operations in Section 5.4.1. Section 5.4.2 then maps these basic operations to those needed for an SVS index. Finally, we propose a novel pruning-based approximation technique to largely reduce the computation overhead incurred by OMD computation in Section 5.4.3, which is our unique system contribution when dealing with clustering in the OMD space.

5.4.1 Basic data structure and operations

SVS organization. Taking a leaf out of previous work [230], we organize SVSs with a tree. Each leaf node in this tree represents a unique SVS. The root node represents all SVSs stored in the index. Each internal node is the root of a subtree and represents all SVSs at the leaves in this subtree. Each children node of the same parent node contains a subset of SVSs of that in parent node. As clustering SVSs can be seen as partitioning a set of SVSs into multiple subsets, the SVS tree structure encodes multiple ways of clustering SVSs.

Evaluating cluster tree using dendrogram purity. The quality of a cluster tree is less obvious compared to a "flat" clustering approach, such as k-means. We adopt a holistic measure, known as *dendrogram purity* [231]. In words, the dendrogram purity of a cluster tree with respect to a ground truth clustering C^* is the expectation of the random



(a) Initially (b) C_0 masks T_0 (c) Unmask T_0

Figure 5.7: Masking and unmasking

process computed in the following steps: 1) sample two SVSs, s_i and s_j uniformly from the same cluster C_x in the ground truth; 2) compute their least common ancestor p in the cluster tree T; and 3) compute the fraction of leaves of p that also belong to C_x. With this definition, the dendrogram purity is a real number between 0 and 1. A "perfect" cluster tree has dendrogram purity 1, in which all leaves of an internal node belong to the same ground-truth cluster.

Greedy incremental clustering. For live video feeds, new SVSs arrive continuously, hence it is natural to follow a greedy incremental insertion strategy. For each incoming SVS x, the algorithm will iterate through all the leaf nodes in the cluster tree to find the nearest neighbor to x, a leaf node s. A `Split` operation is performed on s that first disconnects s from its parent, creates a new leaf node s' to store x, and finally creates a new internal node, whose parent is s's former parent and whose children are s and s'.

Masking and unmasking. However, this greedy insertion strategy cannot reach an optimal dendrogram purity because of the *masking effect*. Figure 5.7 illustrates the case when the current leaves in a tree belong to a 'car-dominant" cluster, denoted C_0 and C_1. If an incoming SVS T_0 belongs to a "train-dominant" cluster, according to the greedy incremental tree construction heuristics, it will be first inserted in the tree next to C_1. We say T_0 is masked by C_0, since C_0 and C_1 are clearly more similar to each other and should be sibling nodes. Masking effect essentially means putting a SVS is being covered by a supposed representative that does not actually represent it well. The root cause of masking effect is that

Algorithm 1 Rotate(v, conditionDetect)

(ShouldRotate, ShouldStop) = conditionDetect(v)
if ShouldRotate **then**
  $Tree$.Rotate(v, conditionDetect)
end if
if ShouldStop **then**
  Output: $Tree$
else
  Output: $Tree$.Rotate(v.Parent(), conditionDetect)
end if

the random incoming order of SVSs cannot guarantee siblings in a cluster tree are actually the closest (like T_0 and C_1 in Figure 5.7b).

Formally, A node v with sibling v' and aunt a (i.e., the sibling of a parent node) in a tree is masked if there exists a point $x \in lvs(v)$ such that, $\max_{y \in lvs(v')} OMD(x, y) > \min_{z \in lvs(a)} OMD(x, z)$. $lvs(z)$ is the set of leaves for any internal node z. [230] proves that a cluster tree without masked nodes has dendrogram purity 1.

Inspired by self-balancing trees like red-black trees [232], we employ the following masking triggered rotations to mitigate the dendrogram purity loss. When masking is detected, a `Rotate` operation swaps the position of s with its aunt in the cluster tree. After the rotation, the algorithm checks if s's new sibling is masked and recursively applies rotations until no masking is detected or we reach the root (Algorithm 1). The intuition behind the rotations is to move the "masked" node up towards the root by swapping this node with its aunt recursively, until masking no longer manifests.

5.4.2 Building an SVS index

SVS insertion. The combination of greedy incremental insertion and masking-triggered rotation provides a baseline algorithm to insert an incoming SVS (Algorithm 2). This is referred to as purity enhancing rotations for cluster hierarchies (PERCH [230]), building a cluster tree with optimal dendrogram purity. Further, the algorithm also performs a balancing-triggered rotation. This is an optimization we will discuss in the following subsection, together with a pruning-based nearest neighbor search optimization.

SVS clustering. Given the SVS tree, we derive the actual clusters using the following

heuristics. We maintain a list of tree nodes to represent different clusters. Initially, only the root node of a tree is in this list. While the number of nodes in the list (i.e., the number of clusters) is less than k (determined by the silhouette method mentioned in Section 5.3.3), we iteratively remove the tree node with the smallest *cost* in the list, replacing it with its two children nodes added to the list. The clustering process terminates once there are exactly k tree nodes in the list. The *cost* of a tree node v is defined as the maximum distance among the $lvs(v)$.

Search operations. Our index supports two basic search primitives: *feature search* and *SVS search*, corresponding to the direct identification query and clustering query. A *feature search* aims at finding a set of SVSs that might contain feature vectors that are similar to the target feature vector. We use the decision-boundary-based query (Section 5.3.3), first to identify candidate clusters of SVSs, and then searching all SVSs in candidate clusters to find the SVSs that actually meet the requirement. An *SVS search* aims at finding the SVS that is closest to the target SVS. This is supported by the nearest neighbor search function in the insertion algorithm.

Smoothness of the OMD space. The latent space generated by SVSs may not be smooth, especially when an SVS consists of high-dimensional feature vectors. Due to the *curse of dimensionality* [233, 234], we cannot avoid this issue completely. This "discontinuity" causes some SVSs to be inserted into the wrong clusters, and a subsequent query to *Video-zilla* may either miss an SVS (false negative) or examine an SVS unnecessarily (false positive).

Given the nature of video scenes, we cannot derive close-form expressions for this discontinuity. To quantify its impact empirically, we evaluate the false-positive (FPR) and false-negative rates (FNR) of queries with and without *Video-zilla* in Section 5.7.4. Further, we also introduce a performance adaptation and bailout mechanism in Section 5.5.3.

5.4.3 Nearest neighbor search optimization

One sub-procedure of the algorithm in 5.4.1, finding nearest neighbors in the cluster tree, can make its naive implementation slow. In this section, we introduce pruning-based nearest neighbor search with the help of an easy-to-compute lower bound of OMD. In addition, we

perform a balancing based rotation operation as further optimization.

Pruning-based search. We propose a novel pruning technique by introducing a cheap lower bound of the OMD that allows us to prune away the majority of SVSs in the cluster tree without ever computing the exact OMD distances.

Following [227], it is straightforward to show that the distance among the centroids of feature vectors within two SVSs is a lower bound for the OMD between them. We refer to this lower bound as the object centroid distance (OCD) as each SVS is represented by the weighted average vector over all feature vectors contained in it. Compared to OMD, OCD is easy to maintain and update in $O(N)$ time, where N is the feature vector dimension.

We use OCD to drastically reduce the amount of OMD distance computation when performing the nearest neighbor search. We first sort all SVSs within the cluster tree in ascending order of their OCD distance to the target SVS. Starting from the head of the sorted list, we repeatedly compute the OMD between the target SVS and the SVS that is at the head of the sorted list. The distance of this SVS will be updated and the list will be sorted based on the updated distance. As the OCD is a lower bound of OMD, we maintain a set of explored SVSs whose OMD has been computed. The first SVS we visited that has already been explored is guaranteed to be the nearest neighbor of the target SVS.

Although pruning-assisted process can largely reduce the computation overhead of nearest neighbor search, it cannot reduce the OMD computation when checking the masking condition or updating the *cost* for each internal node. As the masking condition is checked in a bottom-up manner and the check terminates when no further masking effect is detected, only a small subset of the SVSs within the index are involved. For the *cost* update for each internal node in the tree, we adopt a bottom-up approximation heuristic. The algorithm updates the *cost* of the nodes along the path from the leaf to the root, terminating once the *cost* of a node within the path does not change.

Balancing based rotations. Our pruning method can be largely affected by the depth of the tree. While our rotation algorithm guarantees optimal dendrogram purity, it does not provide any guarantees on the depth of the generated cluster tree. The balance of a cluster tree T is the average local balance of all nodes in T, where the local balance of a node v with children v_l, v_r, is $bal(v) = \frac{min\{||lvs(v_l)||,||lvs(v_r)||\}}{max\{||lvs(v_l)||,||lvs(v_r)||\}}$. Similar to the masking condition,

Algorithm 2 Insert(x_i, $Tree$)

$nbrs$ = NearestNeighbor(x_i)
$leaf$ = Split($nbrs$)
for a in Ancestors(l) **do**
    a.AddPt(x_i)
end for
$Tree$ = $Tree$.Rotate($leaf$.Sibling(), CheckMasked)
$Tree$ = $Tree$.Rotate($leaf$.Sibling(), CheckBalanced)
Output: $Tree$

we will rotate the current node with its aunt in the tree if we find that it will improve the balance of the tree without causing masking.

5.5 *Video-zilla* system

Based on the clustering mechanisms described previously, *Video-zilla* (Figure 5.1) builds a hierarchical index with two components: 1) an *intra*-camera index per camera feed to index the video streams captured by the same camera; 2) an *inter*-camera index across all cameras to index the representative semantic video streams constructed by all intra-camera indices. The *inter-camera* index resides somewhere centrally, where the queries are issued, while the *intra-camera* indices could stay at edge servers to filter raw data and ensure most of them remain local. Each camera sends raw video data to the nearest edge server, via wired or wireless connections. The hierarchical index maintains the boundary between cameras, which allows the components to map to different processing locations when appropriate.

Video-zilla is designed as a layer between the video store and applications. Consider Microsoft's Project Rocket [235], for example. *Video-zilla* can be interposed between the light and heavy DNN detectors, using the feature vectors extracted by the light DNN detectors to build the index, and the heavy DNN detectors to refine the final query results.

5.5.1 Hierarchical index generation

An incoming image frame will first pass a key frame selection module, which adaptively filters video frames based on the computation capacity of the edge server. The selected key frames will then pass several feature extraction modules to compute application-specific

feature vectors. Our automatic video segmentation module will then segment a stream of feature vectors generated by the same camera into separate SVSs. These SVSs will be first inserted into the corresponding intra-camera index. Finally, triggered by the representative SVS update in an intra-camera index, the per-model inter-camera index will be updated. Following this process, our hierarchical index will be constructed incrementally.

Automatic video segmentation. Given a stream of feature vectors, the first key step is to derive an SVS from them. This can be especially helpful for scenarios with frequent scene changes, e.g., as a drone flies above different terrains. We propose a greedy segmentation heuristic. The general idea is to track novel features in consecutive frames and then segment the video when a set of features seems to drift away from the previous SVS.

Initialization. To bootstrap the system, the initial portion of the streaming video over a set length of time t_{max} is extracted as the first SVS. This t_{max} also serves as the maximum length of an SVS. The choice of t_{max} will affect the query result granularity. In practice, we leave the choice of t_{max} to the application developer. In our evaluation, we set t_{max} as 15 minutes.

Tracking novel features. After initialization, we use the representative SVS of the cluster that the last segmented SVS belongs to as the reference SVS. We maintain a current feature buffer that contains all incoming feature vectors after the last video segmentation. An incoming feature vector that lies outside the decision boundary (Section 5.3.3) of the representative will be labeled as a novel feature. We track all novel feature vectors using a novelty feature buffer.

Video segmentation. Whenever a novel feature vector is added to the novel feature buffer, we cluster feature vectors in the buffer using k-means and calculate the average distance between the members and the corresponding cluster center. We compare this average distance d_n in the novelty buffer with d_r in the representative. Video segmentation (Algorithm 3) is triggered when $d_n \leq d_r$. At the same time, we also record the hit pattern of the weighted clustering centers in the reference SVS. if one of these centers has not been hit by any incoming features after time t_{split}, the video needs splitting. t_{split} is set to be $\frac{1}{10}t_{max}$ empirically. To segment the video, the current feature buffer is divided at the point where the first novelty feature or the last hit feature arrives.

Algorithm 3 Video segmentation

```
% A novel incoming feature vector v_novel
NovelFeatureBuffer.add(v_novel)
CurrentFeatureBuffer.add(v_novel)
d_n = NovelFeatureBuffer.calAvgDist()
d_r = SVSTree.avgRepDist()
t_hit = SVSTree.maxLastHitTime()
if d_n ≤ d_r or t_hit > t_split then
    SVS_new = Features(CurrentFeatureBuffer)
    insert(SVS_new, SVSTree)
end if
```

Take a train-station surveillance camera as an example. It should mainly see trains and people (with luggage) or a largely empty platform. The video feed captured by this camera will be divided into chunks using our automatic video segmentation mechanism, correlating with train arrival and departure.

Hierarchical index update. The newly generated SVS is inserted into the corresponding intra-camera index. One or more representative SVSs in this intra-camera index will be updated. The updated representative SVSs will then replace the outdated versions in the inter-camera index. This completes one update round of our intra- and inter-camera hierarchical indices.

Adaptive key frame selection. It is neither computationally feasible nor efficient to process every single video frame captured by a video camera. This module determines which video frames should have objects clipped and passed to the feature extraction module. The ingestion computation cost is determined by both the frame rate and the inter-frame deviation threshold. The latter one uses the deviation between two consecutive images as the metric. When the deviation exceeds a threshold t, we consider the incoming image as a key frame. Many combinations of these two factors are plausible, and we adopt a best-effort heuristic to avoid queuing. We monitor the input frame queue at the feature extraction module. Once a queue starts building up, we will downgrade it to a more lightweight configuration. Conversely, we will upgrade it to a more heavyweight configuration.

Customizable feature extraction. As different applications may prefer specific feature extraction techniques, we make this module customizable. Each application can register their own feature extraction module. We provide several default feature extractors, including VGG16, VGG19, ResNet50, and ResNet101 [236], all trained on the COCO

dataset [124].

5.5.2 Query processing

For direct object identification, the query specification includes an image containing the object of interest, together with the optional time range and camera ID constraints as the metadata. The candidate representatives SVSs will be first identified in the inter-camera index. Then the query will be dispatched to all the intra-camera indices containing the candidate representatives, to search for the actual SVSs that contain the queried object.

A clustering query takes as input a feature map that characterizes an SVS, as well as the optional time range and camera ID constraints. For this query, *Video-zilla* returns all similar SVSs to the input SVS. *Video-zilla* will check the inter-camera index to find the cluster C containing the most similar SVS to the input SVS. Each intra-camera index will return all SVSs that belong to the cluster represented by a representative SVS in C.

5.5.3 Performance monitoring and bailout

Performance monitoring. Due to the errors induced by feature extraction, index building and SVS candidate selection, querying our hierarchical index cannot guarantee 100% accuracy. Thus, *Video-zilla* monitors the error rate of query results and adjusts the index setting to satisfy the user-defined error preference. Periodically, in addition to querying the hierarchical index, *Video-zilla* runs the same query on all the video frames in the background. This query result serves as the ground-truth. This will inevitably take a long time, so *Video-zilla* only performs this operation every 50 queries.

If the current query F1 scores do not meet the user preference, *Video-zilla* can adjust the following parameters one at a time: i) increasing the number of clusters within the inter- and intra-indices; ii) decreasing the OMD computation threshold to derive a more accurate OMD value; and iii) downgrading to a flat SVS index, i.e., without distinguishing between the intra- or inter-camera indices.

Bailout. If applying all three parameter adjustments still cannot meet the user error preference, a bailout mechanism will be triggered, where *Video-zilla* will downgrade to a frame-level index to search through video frames across all cameras. Meanwhile, *Video-zilla*

periodically runs a query on the hierarchical index to determine when to switch back to the hierarchical index. *Video-zilla* performs this operation every 10 queries.

5.5.4 Discussion

Privacy considerations. Orthogonal to common privacy protection approaches such as data encryption and image anonymization (via pixelation or blurring), *Video-zilla* introduces some amount of anonymization through feature aggregation. Given our hierarchical index design, only the representative SVSs in each intra-camera index will be sent to a centralized operation point to construct an inter-camera index. An analytics user can only access the few frames returned as the query results and not the raw videos at large.

Security concerns. IoT cameras have been vulnerable to security attacks or exploited in DDoS attacks (such as Mirai [237]). While this work does not explicitly address security issues, *Video-zilla* may help by acting as an intermediary, which facilitates limiting direct access to the cameras.

Per-model indexing. *Video-zilla* generates an index per DNN model. However, the index generation process is the same, regardless of the feature extractor, and hence different applications can run on a common indexing layer.

Camera ID and time range filtering. We can filter the camera ID at the *inter*-camera level when identifying the *intra*-camera indices to dispatch the queries to. The time range can be applied in each intra-camera index, by comparing against the video timestamps.

5.6 Implementation

We implement *Video-zilla* using the Akka toolkit [238]. Following the actor model in Akka, *Video-zilla* comprises of five actors, key frame selection, feature extraction, intra-camera index, inter-camera index and query history cache. Data exchange between modules follows asynchronous message passing. There is also a query actor per type of queries.

We use the OpenCV library [18] to implement key frame selection, including video I/O, frame rate control and image deviation computation. Keras 2.3.1 [239] is used to train the default feature extractors, and the massive online analysis library [240] for representative

construction. The fastOMD implementation adapts the fastEMD library [241]. Our incremental clustering algorithm extends the codebase in [230] to our OMD metric space. Each intra-/inter-camera index is stored as a tree structure.

Our feature extractors are trained using Microsoft COCO, and this code is written in Python. Other *Video-zilla* modules are written in Java. The implementation totals about 4K lines of code, including 3K in Java and 400 in Python for the indexing, and around 500-600 lines of Java code for the query stubs. The codebase size of *Video-zilla* suggests that existing analytics systems can be simplified substantially by issuing queries to a generic index in place of performing custom video content analysis per application.

Setup and configuration APIs. The analytics application can add or remove a feed with `cameraStart(cameraID, historyDataTimeRange, appID)` and `cameraTerminate(cameraID, appID)`. We make the feature extraction module pluggable and expose an API to applications to customize feature extractors. `setFeatureExtractors(Model, appID)` allows an application to specify a custom feature extractor to characterize the video semantics.

Query APIs. *Video-zilla* supports two most common queries, `directQuery(objectImg, appID)` for direct object identification, and `clusteringQuery(targetSVS, appID)` to organize SVSs. A direct query takes as argument the image of the object of interest, while a clustering query takes as input the feature map of an SVS. We further provide a utility API, `getMetaData(SVS)`, to return information such as the start and end timestamps of this SVS, the camera ID that captured this SVS, and the access time of this SVS.

Customizable APIs: Video archiving as a case study. Developers can also implement custom APIs by composing the built-in APIs. For example, to build a video archiving service, we need an API `isArchived(targetSVS, appID)`, which is a variation of the clustering query. Instead of returning a list of semantically similar SVSs, it returns the average access frequencies of all these SVSs. Code snippet 5.6 shows how to implement this by composing `clusteringQuery` and `getMetaData`.

```
isArchived(targetSVS, appID):
    SVS[] res = clusteringQuery(targetSVS, appID);
    sumFreq = 0;
    for (SVS svs : res):
```

```
        sumFreq += getMetaData(SVS).accessFreq;
    return sumFreq / res.length;
```

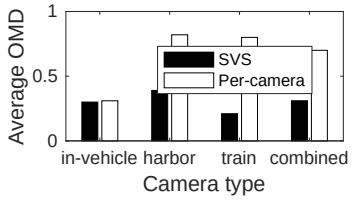
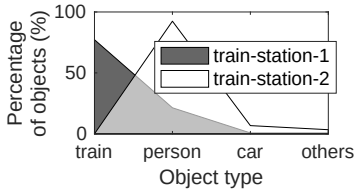
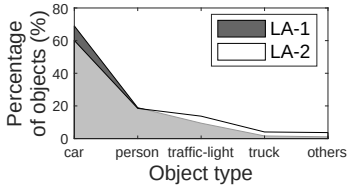


Figure 5.8: OMD comparison.



(a) Train-station camera



(b) In-vehicle camera in LA

Figure 5.9: Object distributions from the same feed.

5.7 Evaluation

Hardware setting. We use two Linux servers to host the hierarchical index, both with 8-core 2.1 GHz Intel Xeon CPUs, one with a NVIDIA RTX 2080Ti GPU, and the other with an NVIDIA RTX 2070 GPU. We run an inter-camera index on the former and intra-camera indices on the latter. In cases we need to run multiple intra-camera indices, this process is done in a serial fashion on a single machine. This way we avoid any artifacts that may arise from a distributed setup. *Video-zilla* is orthogonal to improving processing efficiency with distributed computing, and can leverage existing works [36, 242, 243] on that.

Datasets. For the microbenchmarks (Sections 5.7.1, 5.7.2, 5.7.3), we synthesize a dataset from 1000 SVSs. Each contains 500 1024-dimension feature vectors, and these vectors follow a multivariate normal distribution. We assume there are 10 different types of feature vector distributions in total, each containing 100 SVSs.

For the end-to-end case studies (Sections 5.7.4, 5.7.5, 5.7.6), we assemble publicly available real-world datasets to emulate a real-world multi-camera surveillance system for public transportation, like that in Chicago [125]. The total length of these videos is 30 hours, including three types of feeds from both stationary and moving cameras: i) 40 road-view captured by in-vehicle cameras: 20 of them capture the downtown areas [126] of New York City, London, Chicago and Los Angeles, 5 per city; the other 20 capture highways across the U.S. [127]; ii) 2 train-station video livestreams from Youtube [128, 244]; and iii) 2 harbor video feeds from Youtube [129, 245]. We intentionally set the number of cameras in

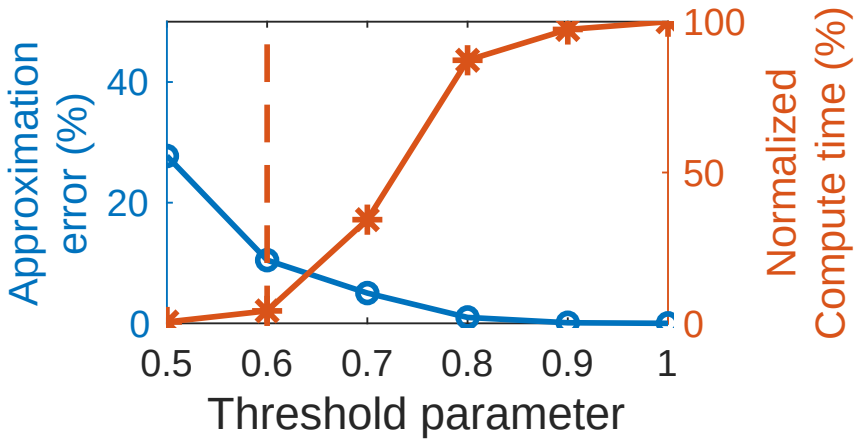


Figure 5.10: The impact of threshold on FastOMD.

train-stations and harbors to be smaller than that of in-vehicle cameras, which matches the expected ratio between these feeds in practice.

5.7.1 Comparison of video characterization

SVS vs. Camera-level. We compare the SVSs from the same camera feed with those belonging to the same semantic cluster (which may or may not be from the same camera). For both sets of SVSs, we compute the average OMD distance between the SVSs pairwise. Besides the first three real-world scenarios, we derive a combined case by concatenating a downtown road-view video with a highway-to-national park road-view feed. This emulates a car driving from a downtown area to a highway, to increase the variability of content.

We use four types of feeds, including 10 in-city and 10 on-highway camera feeds as the in-vehicle case, 2 "harbor" feeds, 2 "train-station" feeds, and 10 camera feeds as the last combined case. The average length of SVSs extracted by *Video-zilla* is around 10 minutes worth of video, 600-700 frames.

Figure 5.8 shows that the average OMDs of in-vehicle cameras are similar to one other, as the feeds we captured are relatively short and the frame content within each feed is fairly homogeneous. Empirically, the OMD among the SVSs within the same *cluster* is around 0.3 and 0.35. However, there is a distinct difference between SVSs in the other three cases. A lower average OMD for *Video-zilla* means that *Video-zilla* is able to better capture the semantic similarity compared to camera-level video characterization. Further, it shows the OMD calculation works regardless of whether the video frames are from stationary train-

station cameras or moving in-vehicle cameras, as the object detection mechanism *Video-zilla* adopts works for both cases. As an example, Figure 5.9a shows object distributions from two SVSs derived from a train station surveillance camera, and Figure 5.9b shows the same for an in-vehicle camera in LA. The object distribution over time hardly changes for the LA road-view feed. In contrast, for the train-station feed, the object distribution varies depending on what events are covered. This highlights the descriptiveness of an SVS over a camera-level video feed.

SVS vs. Frame-level. Using frame-level characterization results in many more data objects to handle. Figure 5.9 suggests that SVSs as data units for queries capture the object distribution, SVS can improve system scalability without sacrificing descriptiveness.

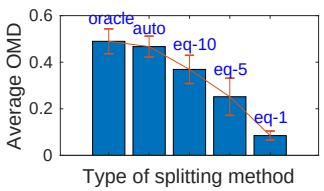
5.7.2 Effective and efficient SVS derivation

Automatic video segmentation. We select and concatenate 10 feature maps as SVSs from our synthesized dataset. To mimic real settings where SVSs differ in length, the number of composite feature vectors within each SVS is a random number ranging from 250 to 750, and the feature vectors in each SVS follow a distinct multivariate normal distribution.

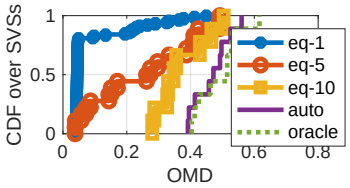
Our automatic video segmentation technique is compared with two baseline methods: (i) an oracle that can perfectly segment a video feed, which, in our case, means segmenting the video feed exactly into the 10 original SVSs; and (ii) a strawman that simply divides a video stream into equal-length video clips, where the fixed length can be 1, 5 or 10 minutes' worth of video clips.

We evaluate the effectiveness of the video segmentation techniques using the average OMD distance between SVSs that appear consecutively in a feed. Higher OMDs mean better segmentation effect. Figure 5.11a shows the average OMD distance using different segmentation techniques. Our method nearly matches the oracle while outperforming the strawman method. Zooming in, Figure 5.11b shows the empirical CDF of the OMD between adjacent SVSs. When video clips are segmented evenly, it is unavoidable that some adjacent clips are similar to each other and exhibit low OMDs.

Fast OMD. Figure 5.10 shows the impact of the FastOMD threshold α on the computation accuracy and time. We randomly select 100 pairs of SVSs from the synthesized dataset,



(a) Average OMD



(b) OMD distribution

Figure 5.11: OMD between adjacent SVSs

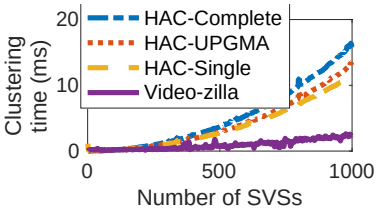


Figure 5.12: Clustering algorithm comparison.

and compute the pairwise OMD distance as α varies from 0.5 to 1. We use the OMD for $\alpha = 1$ as the reference ground truth and calculate the approximation error as the accuracy metric. The computation time is normalized with respect to the computation time needed for $\alpha = 1$.

The approximation error decreases with increasing α at the expense of a higher computation time. Empirically, $\alpha = 0.6$ appears to achieve a balance between the computation accuracy and processing time reduction. For this α value, the average OMD computation time is reduced to less than a second (767 ms on average). Since the average length of the SVSs in the real-world dataset is around 12 mins, this overhead is acceptable.

5.7.3 Scalable incremental clustering

Pruned nearest neighbor search. Figures 5.13a and 5.13b show computation reduction when inserting or querying an SVS given different index sizes, measured by the number of SVSs with the index. For SVS query, the nearest neighbor search is the only trigger for OMD computation, and our pruning approach can reduce the computation by 92%. For SVS insertion, checking for masking and updating the cost of internal nodes incur additional OMD calculations, but pruning still reduces the total OMD computation by 80%.

Incremental SVS clustering. Figure 5.12 compares Video-zilla's incremental clustering algorithm with the commonly-used hierarchical agglomerative clustering (HAC) algorithms [246] with differing linkage choices. All algorithms achieve similar clustering accuracy. Every incoming SVS triggers a clustering attempt to update the representative SVSs of the index. The overhead of the three HAC algorithms increases quadratically with the size of the index, while that of *Video-zilla* only linearly by avoiding reconstructing the whole tree every time.

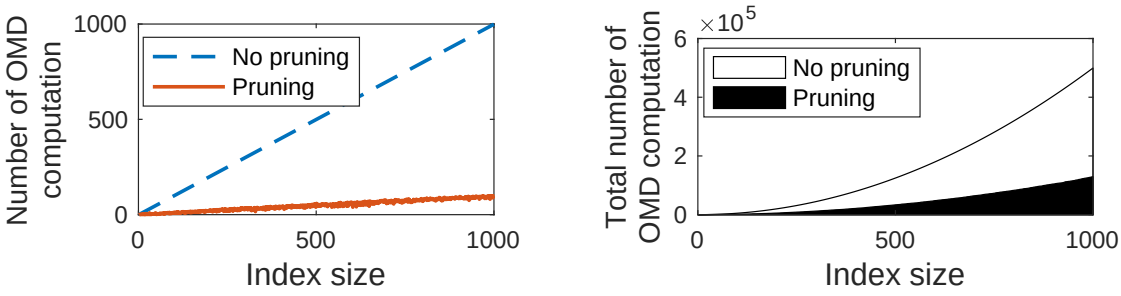
Comparison with M-tree. SVS organization leverages a hierarchical incremental clustering tree, which is a well-studied area. We compare our method (PERCH with OMD approximation, or "PERCH-OMD") with M-Tree [247], an efficient incremental indexing method for similarity search in a metric space. Using the synthesized dataset, we perform a 100-nearest-SVS search for 10 randomly selected SVSs. We choose 100 because this is the ground-truth cluster size for each SVS cluster in the synthesized dataset.

In Figure 5.14, the x-axis represents the maximum number of elements in a single node of an M-Tree, which we refer to as the maximum node size, and the y-axis represents the number of OMD computation needed. For comparison, the dashed line shows the number of OMD computation needed when using PERCH-OMD.

Both PERCH-OMD and M-tree can derive the correct set of 100 nearest SVSs, but M-tree incurs extra OMD computation. This shows that the choice of the maximum node size can heavily influence the number of OMD computation needed. Further, this choice depends on the number of SVSs within a cluster, which varies with the actual video feeds. The main reason is that data structures like M-tree still suffer from the masking effect mentioned in Section 5.4. Elements which belong to the same SVS cluster can be contained in disjoint subtrees, while some SVSs that should belong to different clusters are included in the same subtree. The extra OMD computation arises from comparing with these extra elements. The original M-tree algorithm may also suffer from potential overlap between different leaf nodes. However, this overlap problem appears to be addressed in most open-source M-tree implementations.

Comparison with approximate nearest neighbor (ANN) search. Since PERCH-OMD performs *precise* nearest neighbor search, we also compare it with its approximate counterparts. Specifically, we compare with a state-of-the-art ANN algorithm [248]. This is one of the most efficient ANN algorithms [249], with a well-documented open-source toolkit [250], including built-in support for the EMD metric space.

As before, we perform a 100-nearest-SVS search for 10 randomly selected SVSs in the synthesized dataset. The average recall for this ANN algorithm is 97.8%, which is even slightly better than reported in the original paper (92.5%). This is due to the difference in the dataset. Still, we can observe accuracy loss when applying ANN. In contrast, PERCH-



(a) Per SVS query

(b) Per SVS insertion

Figure 5.13: Number of OMD computation.

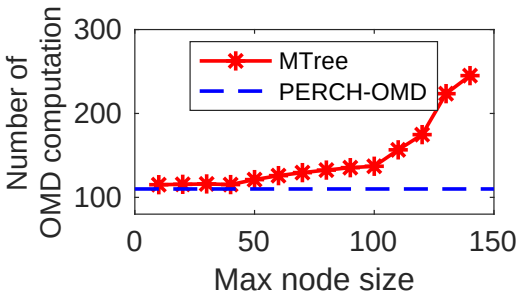


Figure 5.14: PERCH vs M-tree

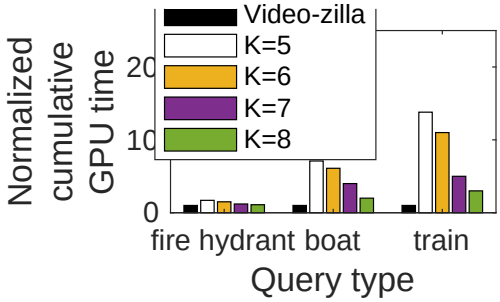


Figure 5.15: The impact of K

OMD considers strictly nearest neighbors, which are by nature more accurate than ANN. Efficiency-wise, given we are not dealing with a huge number of SVSs, we do not necessarily need the "most computationally efficient" algorithm.

Index building overhead. For an index that contains 1000 SVSs in our synthesized dataset, the overall index size is less than 5 MB. The corresponding video length is around 200 hours, whose data size can be more than 20 GB if the video data are captured in 640 × 480 pixels. The overall index building time is less than 20 minutes, which is negligible compared to the total length of video.

The hierarchical index construction in *Video-zilla* naturally lends to distributed indexing. In contrast, building a single, flat centralized index would send more raw data to a centralized point. We therefore compare the amount of data sent for index construction between these two approaches. Assuming key frame selection and feature extraction done at the edge, we incrementally add 20 camera feeds to the system, each containing 100 SVSs randomly selected from the synthesized dataset. By only sending representative SVSs, adopting a hierarchical index can reduce the network traffic between the cloud and edge servers by a factor of 19.

5.7.4 Case study: Direct object identification

Direct object identification queries follow the format of *find image frames that contain object X within all streams*. We let X be *fire hydrant*, *boat*, or *train*. This is because the object of interest should be contained in some but not all videos. Searching for *giraffe* or *person* would not make sense, as no videos in our datasets contain giraffes and virtually all contain people. We generate 50 query instances, taken from the real-world videos, per query type.

We compare *Video-zilla* with two state-of-the-art correlation analysis mechanisms in large-scale video analytic systems, per-camera top-k indexing in FOCUS [204] and leveraging spatial-temporal correlation between cameras in Spatula [205]. FOCUS optimizes the processing latency of individual camera feed by building an approximate frame-level index per feed at ingestion time to reduce query time later. Spatula speeds up multi-feed analytics by leveraging the insight that objects found in one camera feed will only appear on nearby cameras with spatial-temporal correlation with the first camera. Both systems include additional system optimizations such as CNN compression to reduce ingestion overhead and pruning errors via replay search that are orthogonal to *Video-zilla*, so we focus on comparing the query quality and latency for *Video-zilla* based on SVSs vs using the approximate frame-level index in FOCUS or the spatial-temporal correlation between cameras in Spatula. Specifically, we compare the *intra*-camera processing of *Video-zilla* to using the per-camera top-k index and the *inter*-camera processing of *Video-zilla* to a Spatula-like system leveraging spatial-temporal correlation. Unless otherwise stated, all queries in this section achieve at least 95% precision and recall.

***Video-zilla* vs per-camera top-k.** For each camera, we build an approximate top-k index for the incoming image frames. An incoming query will be directly dispatched to the top-k indices of *all* camera feeds, instead of only to the inter-camera index in *Video-zilla*. To ensure the performance is not affected by unrelated factors, we use the same ResNet50 used in *Video-zilla* feature extractor with the additional `softmax` layer to generate the top-k index, and we use the same set of video frames to build indices for both systems. This way, the ingestion overhead of both systems is roughly the same. As in [204], we set $k = 3$ for all top-k indices, i.e., each object in a streaming video will be indexed by the top 3

possible object classes in the corresponding top-k index. We adopt the same Yolo-v2 as the ground-truth CNN in [204].[1] The ground-truth CNN is the main contributor to the query time.

We measure the bottleneck query time (Figure 5.16), i.e., the time taken for the slowest intra-camera index to return results. This is because the end-to-end query time is bottlenecked by this even when we parallelize query processing of intra-camera indices.

There are two components of the query processing time: one is searching for the right frames, and the other is processing those frames to complete the query. By carefully indexing video data, *Video-zilla* can minimize the search time and thus reduce the total number of frames processed in the latter part, seen in a decrease in the cumulative GPU time, but does not affect the processing time in the slowest intra-camera index. Video-zilla tries to reduce the computation time on unnecessary video data, not to reduce the essential search time.

Figure 5.17 shows the cumulative GPU time across all intra-camera indices. We do not show the cumulative CPU time here as GPU is typically the resource bottleneck for deep learning inference workloads. While achieving nearly the same query time for all types queries, *Video-zilla* reduces the cumulative GPU time by 14× compared to the top-k indices. Viewed differently, *Video-zilla* can support up to 14× more video queries simultaneously while incurring negligible accuracy loss.

The root cause is that a frame-level top-k index does not capture enough feed-level information, whereas the hierarchical index of SVSs in *Video-zilla* does. The top-k index incurs unnecessary computation if it mis-classifies objects, whereas *Video-zilla* is more robust. Careful inspection of the top-k indices for each camera feed suggests that a train object can be found even in the top-3 index of a video captured in downtown Manhattan. An actual query therefore causes the system to unnecessarily search through the Manhattan feed. Figure 5.18 illustrates this mis-classification effect. For the same video segment, *Video-zilla* will identify three object classes, but the top-k index has 4 classes, the fourth being "other". This "other" class is the source of extra, unnecessary computation in a top-

1. We also tried YOLO-v3 and YOLO-v4, and saw little difference in the object detection accuracy for our datasets. However, it is much faster to run v2.

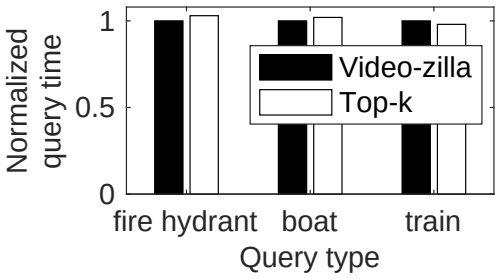


Figure 5.16: Bottleneck query time

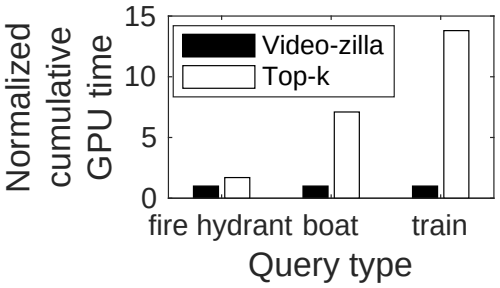
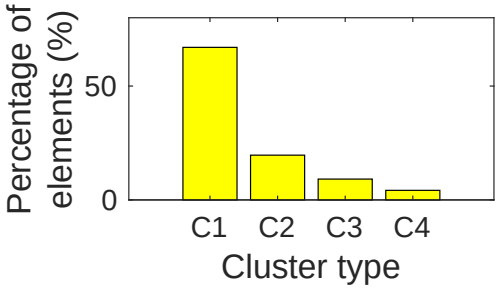
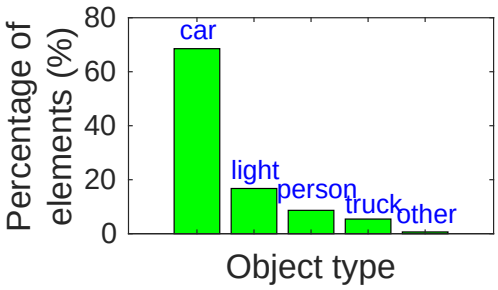


Figure 5.17: Total GPU time



(a) Clusters in *Video-zilla*



(b) Top-k index

Figure 5.18: Feature clusters in the same video

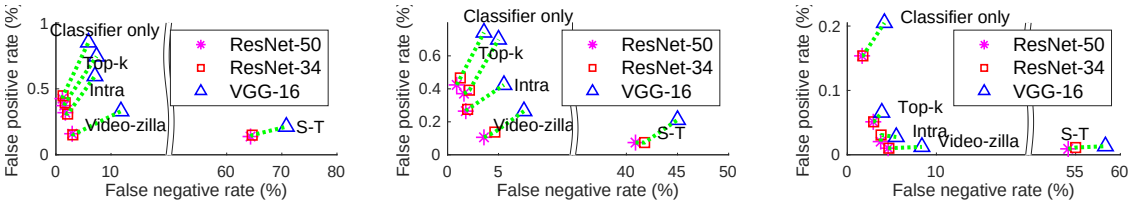
k index, as any video frames associated with this class will be examined during an actual query.

We use K to represent the number of classes that can be recognized in a top-k index. Setting a proper K value can help reduce the mis-classification effect. In our case, we set K to 5. Decreasing K from 5 to a lower value will clearly amplify the mis-classification effect in a top-k index. Therefore we increase K from 5 to 6, 7 and 8. Figure 5.15 shows the impact of the K value on the cumulative GPU time. However, identifying the right K value is non-trivial and requires careful inspection of the specific video feed. The whole point of Video-zilla is to automatically organize data based on its semantics instead. Furthermore, a larger K requires a more complicated recognition model, hence larger processing overhead at ingestion time.

***Video-zilla* vs spatial-temporal correlation.** The main purpose is to show the benefits and caveats of using spatial-temporal correlation compared to the correlation captured in *Video-zilla*. Therefore, we use the same intra-camera query mechanism in *Video-zilla*. For the inter-camera part, instead of dispatching queries based on the SVS similarity, we dispatch queries based on the spatial-temporal correlation between the camera that captured

the image and other cameras within the system. For example, for a fire hydrant query that is captured by an in-vehicle camera in New York City, we will only search other cameras located in NYC.

Both camera-level filtering approaches achieve around 95% query *precision* performance. However, adopting spatial-temporal correlation leads to higher false negative rates (FNRs) (Figure 5.19), suggesting the search space is pruned too aggressively. This is because spatial-temporal correlation is a coarse-grained video similarity.



(a) Fire hydrant (b) Boat (c) Train

Figure 5.19: Performance variation with indexing schemes and feature extractors.

Uncertainty and error rates. Figure 5.19 shows the false positive rates (FPR) and false negative rates (FNR) for various queries. $Recall = 1 - FNR$. We compare indexing schemes including "classifier-only" (no indexing), per-camera top-k, spatial-temporal correlation (S-T), *Video-zilla*, and a version of *Video-zilla* without the inter-camera index, where queries will be sent to all intra-camera indices. Data points corresponding to the same scheme are linked with dash lines. We experiment with three feature extractors.

When using the state-of-the-art feature extractors like ResNet-34 and ResNet-50, *Video-zilla* incurs consistent FNR losses (up to 3%) and up to 80% FPR reduction compared to *classifier only*. The FPR reduction shows *Video-zilla* prunes the search space effectively. S-T achieves similar FPRs with spatial-temporal pruning. However, that pruning is too aggressive, hence dramatically increasing the FNRs.

With the less accurate VGG-16 as the feature extractor, error rates vary by the query goal. The FNR increases for *fire hydrant* because VGG-16 classifies fire hydrants less accurately than it classifies boats and trains, which propagates to inaccurate clustering. Interestingly, *intra only* hardly suffers from increased FNR compared to the top-k index. This suggests a way to mitigate the performance disparity between query types by disabling the

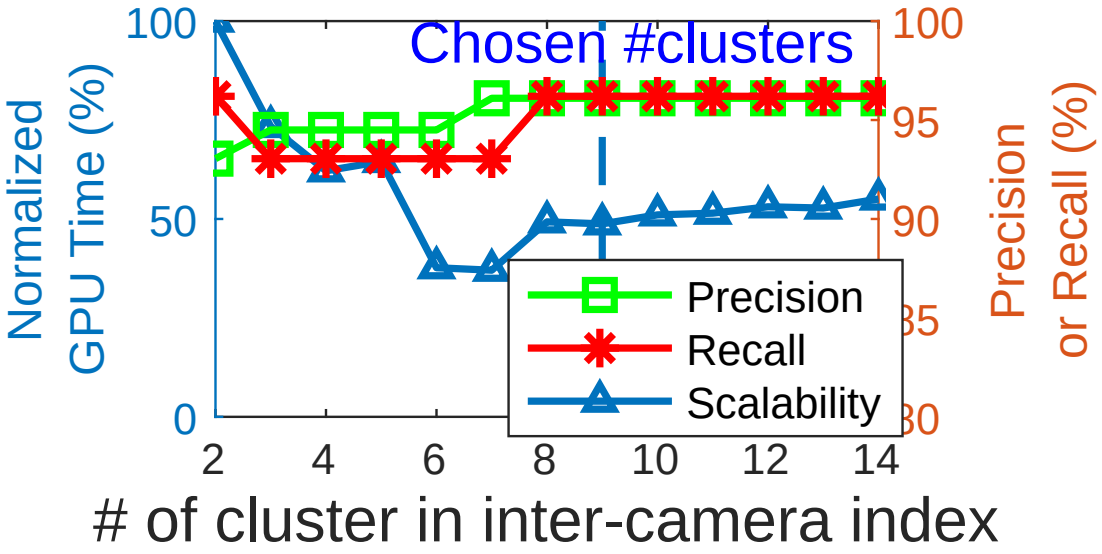


Figure 5.20: Tuning *Video-zilla*.

inter-camera index.

To summarize, the following factors affect the *Video-zilla* performance: First, the FNR is mainly constrained by the feature extractor accuracy; Second, using intra-camera indices only achieves lower FNRs, while a two-level index structure achieves lower FPRs and better processing scalability; Third, certain object types can compound the effect of inaccurate feature extractors, but reducing the number of hierarchies can mitigate the FNR disparity across object types; finally, the threshold of the decision boundary to determine query hit can be adjusted, and wider boundaries typically mean lower FNRs and higher FPRs.

Scalability and accuracy trade-off. Figure 5.20 shows the scalability (in terms of normalized cumulative GPU time), precision and recall trade-off for *fire hydrant* queries as the number of clusters varies within an inter-camera index for *Video-zilla*. *Boat* and *train* queries exhibit similar trade-offs. The red dashed line represents the number of clusters calculated by maximizing the silhouette value on the real-world dataset. We do not vary the cluster number in the intra-camera indices because manually setting that number causes random performance changes in the system. The *precision* generally increases and flattens around the chosen number of clusters. It is in fact convex, and will decrease when there are so many clusters that each SVS forms a distinct cluster. Conversely, *recall* is concave, and is highest when there are very few or many clusters. *Scalability* is similarly concave.

These results suggest that the precision/recall can be tuned by adjusting the number of SVSs clusters within an inter-camera index. More clusters initially increase *precision* and decrease *recall* by aggressively eliminating irrelevant video frames, including some relevant frames with few objects. The aggressive pruning also reduces the GPU time dramatically. As the cluster number increases further, pruning becomes less effective; more SVSs are examined, which increases both *recall* and the GPU time.

5.7.5 Case study: Specialized DNN training

A clustering query follows the format of *find all semantic video streams that are semantically similar to a video stream Y*. In this section, we highlight the potential of this novel query primitive by showcasing its application to specialized neural network training. We compare the training process based on the results of the clustering query compared to leveraging spatial-correlation among cameras. The latter is the state-of-the-art camera correlation model [203]. We use the 20 downtown in-vehicle videos as our dataset, as there is no obvious spatial-temporal correlation among others.

We adopt a method similar to the approach in [72] that only retrains the first and last three layers of a model, given limited size of the training dataset. In our case, we only select k classes of objects that cover 95% of the image frames within the whole dataset. Other classes will be labeled as an "Other" class. Each class uses 500 images as the training data.

We use four models pre-trained on ImageNet in Keras [236] as the base models: MobileNetV2, ResNet50, ResNet101, InceptionV3, which cover a range of accuracy and inference time trade-off. 50 SVSs are selected from our hierarchical index built on top of the 20 downtown videos. For *Video-zilla*, we get a list of video streams that belong to the same cluster, and use them as the training data. For spatial-correlation, we treat the videos captured within the same city as spatially-correlated. Using the output of YoloV2 as the ground truth, we compare the average top-2 classification accuracy for each trained network. *Video-zilla*'s automatic clustering is even slightly better (around 1%) than manual labeling.

By clustering SVSs based on OMD, we successfully cluster video streams that share similar classes of objects and have objects within the same class visually similar to one

other. Both factors can be beneficial when training a specialized neural network. The clustering query by itself can be a novel tool to find the correlation among different video streams.

Manual labeling (tagging feeds with location) is required to leverage spatial-temporal correlation, but *Video-zilla* can automate this process by identifying semantic correlation. Further, manual labeling also implies some manual thresholding. For example, similar cars should appear in cameras at consecutive intersections or 10 exits apart on a highway. These cameras may or may not be labeled as spatially similar, but *Video-zilla* will recognize those.

5.7.6 Case study: Proactive video archiving

By capturing the correlation between different SVSs, *Video-zilla* can enable a proactive video archival service. For example, consider the train-station camera feed mentioned earlier. Most video frames capture empty stations which convey little information. Once a feed is divided into SVS clusters corresponding to "train arrival", "train departure" and "empty station", the "empty station" cluster can be aggressively archived. For any incoming SVS, we can estimate its potential hit rate based on the hit rate of other SVSs in the same cluster, and proactively archive low-information SVSs (for example, to some secondary storage).

For the same query objects considered earlier, we consider the ratio of total temporal length of SVSs that contain each type of queries to the overall video data length. We also take a union of all the SVSs that contain any one of the objects of interest, which emulates the case when an application needs to search for different objects. We assume each object will be queried at the same frequency, since the hit or miss behavior should not change much whether an application query for the same object once or multiple times. What matters is how many different objects are queried, not how often the same (type of) object is queried. The ratio for fire hydrant, boat, train and combined case is 1.5%, 2.0%, 26.3% and 29.1% respectively. Even considering the union query case, less than 1/3 length of video data will be retrieved. This suggests that the storage needs for the video feeds can be reduced by more than 70% by aggressively archiving low-information video segments.

5.8 Related work

We are not aware of previous video indexing work using the collective semantics of an entire video feed. Related work otherwise revolves around video indexing, and video analytics optimizations.

Video indexing and retrieval. There is a rich literature on content-based video indexing and retrieval [251–253], mainly on index design for specific query types, such as *shot boundary detection* [254], *key frame extraction* [255], *semantic search* [256] and spatio-temporal based retrieval [257, 258]. These leverage frame-level or feed-level video data abstractions. *Video-zilla* is orthogonal, since we focus on a new abstraction for video data, SVS, which then lends to an effective index.

Video-zilla can be seen as a form of *explanation database* [206–209] for video data. Index structures have been widely used to reduce query latency in conventional SQL databases [259], key-value stores [260], graph databases [261] and many others [262]. Recent work VStore [263] and VSS [264] aim to design the storage subsystem of a video data management system (VDBMS), but still operate on the raw video data directly. As a novel abstraction to capture the inherent similarity within video data, SVS provides new ways to understand and hence efficiently index video data.

Further, existing semantic video search indices [256, 265, 266] are designed for offline video databases, where optimizations rely on the ability to process the data in multiple passes. In contrast, *Video-zilla* can work for live video feeds, where the main challenges of real-time ingestion and efficient analysis of streaming data arise from large amounts of data continuously arriving and processed in a single pass.

Video analytics systems. Thanks to advances in machine learning, recent video analytics systems have transformed video content analysis from simple information retrieval to sophisticated decision making. Some [267, 268] focus on video data management for emerging applications, like virtual reality and multi-perspective video analytics, while most [36, 242, 243, 263, 269–273] address the inherent video processing complexity by providing system support for video encoding and decoding and parallel and distributed processing. Understanding the semantic meaning of video data is left to upper-level applications.

Recognizing the inherent redundancy within video data, more recent efforts explore smarter video analytics approaches by understanding the video semantics. Noscope [202] and FOCUS [204] both reduce per-feed query cost, the former with cheap per-camera filters and the latter by applying a specialized and compressed DNN per video camera at ingestion. Spatula [205] optimizes cross-camera video analytics by exploiting spatial-temporal locality in a multi-camera deployment, aiming at enabling effective object tracking functionality. In contrast, the notion of semantic video streams (SVS) introduces a new dimension to characterize the correlation among video data and thus provides new optimization opportunities for object identification applications [274]. Further, all these systems implement frame analysis as part of the analytics pipeline, whereas *Video-zilla* acts as a generic indexing layer that can help simplify the development effort for new analytics applications by supporting common indexing operations and query APIs.

5.9 Summary

In this chapter, we propose a notion of *semantic video stream (SVS)* that exposes the semantic content of the video feeds and captures object distribution across a set of frames. This abstraction balances the expressiveness of frame-level analysis and the efficiency of camera-level aggregation. On this basis, we design *Video-zilla*, an indexing layer interposed between a video store and analytics applications. *Video-zilla* builds a hierarchical index to capture of the correlation between SVS instances both within and across camera feeds, so as to dramatically narrow down the search space for common queries. We implement *Video-zilla* as well as proof-of-concept query case studies for object identification, video clustering, and archival. *Video-zilla* lends to almost constant scalability with the number of video feeds. We believe the notion of SVS and the hierarchical index of SVS correlation can also be applied to other domains witnessing large amounts of correlated but unstructured data.

Chapter 6

Linkshare: Device-Centric Control for Concurrent and Continuous Mobile-Cloud Interactions

6.1 Introduction

Mobile applications are becoming increasingly sophisticated, enhancing our interaction with the environment or providing cognitive assistance (Section 6.2). Some scenarios might involve multiple concurrent applications or modules working together (Google Tango [30], Gabriel [31] and DeepEye [32]), where these modules are individually computationally intensive and require computation offloading. Worse, these applications embody a vision of continuous operations, further straining the already limited resources on the mobile devices. These represent canonical examples of multiple concurrent and continuous mobile-cloud/edge sessions.

Although there has been a multitude of computation offloading work over the past fifteen years [12–16, 63–71], existing approaches are limited by intra-application operations only or cloud-centric scheduling.

Instead, the emerging multi-application scenarios can exhibit a heterogeneous network and server execution model, involving different server backends. This suggests different

perceived network latencies and server processing capabilities across applications. Further, they need to share the wireless interfaces, which can only be controlled at the device. Cloud-centric management is no longer sufficient in this concurrent mobile-cloud interaction paradigm. We need to shift the control to a device-driven paradigm. In other words, some operating system level coordination is needed on the mobile side to adequately support the prospects of multi-application offloading.

An analysis of the current mainstream offloading mechanisms suggests that the main consideration is scheduling the network transfer to enable remote computation. In particular, the bottleneck is often the transfer along the first wireless hop. This is important given the advent of mobile edge computing promising to bring the remote server closer. However, the combination of application workloads, network transfer time, and the server processing time complicates the picture. Most of the canonical applications have soft real-time constraints for user interaction. This requires a balancing act between minimizing the end-to-end processing time and meeting deadlines.

For example, the most intuitive approaches of minimizing the end-to-end processing time (Shortest Job First, or SJF) and earliest deadline first (EDF) scheduling both turn out to be inadequate. The former (SJF) gives no guarantee about meeting deadlines and can cause fairness issues (and even starvation). This is because minimizing processing times can penalize a job as a result of *another job* using a less powerful backend. EDF, on the other hand, cannot handle heavy-tailed network transfer time distribution. Section 6.3 analyzes this in detail and suggests adding limited sharing to EDF (EDF-LS). Essentially, when we detect a large network transfer, we serve the next two jobs queued in a round-robin fashion.

Following the above study, we build a system-level scheduler service, LinkShare (Section 6.4) and implement it on Android (Section 6.4.3). LinkShare wraps over the operating system scheduler to coordinate among multiple offloading requests, incorporating the EDF-LS algorithm.

Using benchmark applications representing face recognition, optical character recognition, speech recognition, and license plate recognition, we find that this additional scheduling decision is essential, and EDF-LS reduces the deadline miss rate by up to 30% compared to the baseline EDF (Section 6.5). When the workload is light, EDF-LS hardly incurs penalty

from sharing, and in fact often outperforms EDF. Even if the network bandwidth increases, the deadline concern does not disappear, especially alongside improved server capability.

Although the specific system is motivated by multiple mobile offloading jobs, the issues discussed and the system architecture are generic to concurrent mobile-cloud interactions. In fact, any concurrent, inter-application network-bound requests could benefit from the LinkShare service. More broadly, LinkShare as a framework is also applicable when the "offloading" destination is an edge server or a nearby device instead. It is also amenable to other scheduling algorithms as warranted by the application requirements. We believe LinkShare represents a first step towards coordinating an IoT ecosystem tethered to the mobile device.

In summary, this work studies concurrent and continuous mobile-cloud interactions involving different server backends and argues for a device-centric control mechanism. Specifically, our contribution is three-fold.

First, we analyze the scheduling complexity arising from concurrent offloading workloads. Our study points to the need to balance between meeting application deadlines and avoiding blocking heavy workloads. We adopt limited sharing in addition to Earliest Deadline First.

Second, we build a general framework as a system-level service, LinkShare, that extends the operating system scheduler for concurrent inter-application network-bound requests and incorporates the above scheduling algorithm.

Third, extensive evaluation of an Android implementation of LinkShare confirms the importance of this additional scheduler and shows that EDF-LS achieves the balancing goal.

6.2 Motivation

6.2.1 Emerging Application Scenarios

Mobile Augmented Reality (AR) Games. Mobile AR games have been showing their potential on mobile platforms. One example is the dominoes game included in the Google Tango [30] project. Instead of getting a set of dominoes at hand, users can place virtual

dominoes in the real world and arrange them through their mobile phones. It requires the application to understand the user's surroundings, remember the exact locations of the dominoes already placed, and display the virtual dominoes on the screen in the meantime. To enable all these functions, motion tracking, depth perception, area learning and video rendering modules are needed. The first two modules help with understanding the environment, while area learning helps with remembering the previous operations and video rendering concurrently displays the virtual dominoes. All these modules coordinate with one another to make the dominoes game work correctly.

Wearable Cognitive Assistance. Wearable devices for cognitive assistance have been suggested for more than a decade [275]. More recently, Gabriel [31] provides interactive cognitive assistance using Google Glass to help people suffering from cognitive decline, such as those with Alzheimer's disease. The patients are often unable to remember the names of friends or remember to perform daily tasks. When looking at a person that the user might know, the assistant will tell the user the name of the person immediately. When looking at his/her plants, it will remind him/her to water them. These two scenarios require face recognition and object recognition respectively. As we cannot predict when the user may meet with a friend or walk around his/her garden, these two modules must run continuously and simultaneously.

Smart Video Surveillance. Smart home has become a popular concept in recent years, and smart video surveillance is a key technology to ensure the security in smart homes. According to [276], the key to security is situation awareness. The system needs to keep track of "who are the people in a space?" and "what are the subjects in the space doing?". To answer these questions, learning techniques like face recognition, object recognition, activity tracking are widely employed. To quote one report [277], "one of the biggest factors assisting market growth" is the real-time access and monitoring capability. All the above modules need to run concurrently to generate comprehensive real-time alerts.

Observations. Common across these examples, they are all computation intensive and latency sensitive, and have a continuous flavor. More important, there are multiple modules either within a single application or across multiple applications. Take the face recognition module as an example, the average execution time on Google Glass, a Samsung Galaxy

Android smartphone, and a remote server are 2912, 537, and 41 ms respectively [26]. To maintain a high refresh rate for an interactive face recognition application, it is impractical for all these modules to run entirely on the local device.

6.2.2 Limited Execution Model Currently

The de facto approach to handling computation-intensive mobile applications is to offload computation to the cloud (or, more recently, to the edge). However, the current execution model typically focuses on only one computation-intensive workload running at a time, and most optimizations are applied within a single application. This single-workload model does not suit the emerging applications described above, whether commercialized applications like Google Tango (multiple modules running concurrently within the same application), or research prototypes such as Gabriel (multiple applications running concurrently within a single device), DeepEye [32](multiple deep vision models running concurrently), and MCDNN [72] (multiple deep neural network applications running concurrently, sharing the same library).

Instead, system-level cross-application coordination support is needed to manage simultaneous execution of these applications, but existing approaches are still limited to specific scenarios and do not address concurrently offloading to different backend servers. Tango takes a hardware approach without offloading. MCDNN operates at the DNN library level, aiming to process jobs locally as much as possible. DeepEye limits its use cases to multiple deep vision applications. Gabriel is currently designed for concurrent mobile applications supported by a common backend server and takes a cloud-centric approach.

6.2.3 Towards device-centric scheduling

The entire offloading job involves local processing and/or data transfer to the remote server followed by remote processing. Therefore, we need to consider sharing and scheduling for each component.

With concurrent offloading, different apps may use distinct backend servers, due to either technical or business-related reasons. For an example of the latter, consider the two mainstream public cloud platforms: Amazon EC2 and Google Compute Engine (GCE).

Price-wise, for the same server computation capacity, GCE is cheaper than Amazon EC2. However, the EC2 deployment covers more datacenters around the world (22 total, 7 in the US) compared to GCE (21 total, 6 in the US). Therefore, a cost conscious developer who only needs to provide regional service may opt for a server on GCE, while another developer aiming for a better service coverage worldwide may favor EC2.

We now face a new situation where the control needs to be shifted to the mobile device. Since the concurrent applications and their backend servers are likely heterogeneous, such device-driven control really matters.

Complicating the picture even further, applications have different service requirements, manifested in deadlines for real-time user interactions [278]. On-device scheduling can make a huge difference. Even a simple reordering of the offloading requests could dramatically improve the performance of one module without hurting the others. We will study specific examples next.

6.3 Scheduling offloading jobs

6.3.1 What to schedule on-device

The anatomy of an offloading job. The processing of a job involving offloading includes several components: potentially some local processing on the mobile device, transferring the data to the cloud, and processing on the remote server(s). The second component can be further split into transfer on the wireless access link and on the wired path to the server. The split between local and remote processing is specific to the offloading mechanism adopted in the application.

Within these components, the first component is traditionally managed by the operating system scheduler and access to the wireless link is also initiated by the mobile device. The remaining components, transfer on the wired path and the server processing, are beyond the control of the mobile device. Therefore, the main scheduling decision in our context concerns sharing the wireless link between concurrent offloading workloads.

Offloading mechanism and scheduling. Existing main-stream offloading mechanisms are method-based [12, 13] or, similarly, based on small execution units [16]. In other words,

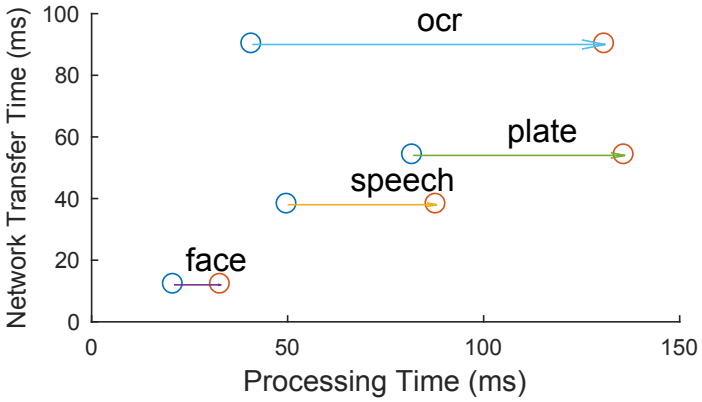


Figure 6.1: Processing Time Breakdown

the whole application process can be divided into many execution units, and each unit is run either locally or remotely.

Consider face recognition. The recognition function is treated as a standalone method in MAUI [12]. Invoking this method on a single frame from the video feed comprises an execution unit. When this application operates on a video feed, each frame is processed in its entirety either locally or remotely. Therefore, we can decouple the decision between local vs remote and how to order the remote ones.

To summarize, an offloading scheduler needs to make two decisions: how many jobs to process locally (then leave to the OS to schedule), and how to order the offloading jobs. Since the first problem has been studied extensively [63–70], we focus on the second problem in this paper.

6.3.2 The complexity of offloading

The offloading performance is affected by multiple factors. The server capability determines the remote processing time. The server's physical location along with the network conditions affect the data transfer time. In particular, wireless links are susceptible to interference and multipath fading.

Figure 6.1 shows the processing time breakdown of our benchmark applications, assuming a 10 Mbps network upload speed. For each application, the point on the left shows the remote processing time, while the point on the right shows the end-to-end processing time (i.e., with data transfer time added to the remote processing time). The timing results are averaged over 500 runs per application. The application and experiment setups are detailed

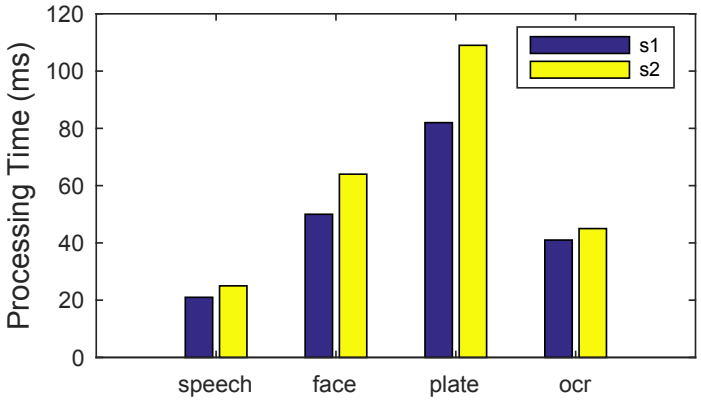


Figure 6.2: The impact of server capacity

in Section 6.5.1.

We can see that plate recognition is clearly computation bound, whereas optical character recognition (OCR) is network bound. While face recognition, plate recognition and OCR are expected to operate on the same image frame[1], face recognition does not need the entire frame, locally or remotely. The built-in face detection function in the vision library can narrow down the region of interest before handing the recognition application a subset of the frame containing these regions only. Therefore, the input size for face recognition is much smaller than the entire frame.

We then measure the processing times on two different servers, S_1 and S_2. S_1 has an Intel Core i5-4570 processor (Quad Core, 6 MB Cache, 3.2 GHz) and 8 GB 1600 MHz DDR3 Memory. This is the same server used for the previous figure and in our evaluation throughout. S_2 is a workstation with an Intel Core i5-4430 Processor (Quad Core, 6 MB Cache, 3.0 GHz) and 16 GB 1600 MHz DDR3 Memory. We randomly select 500 data samples from the input dataset and Figure 6.2 shows the processing times of the benchmark applications on the two servers.

Although the server specifications appear similar, the processing time for face recognition and plate recognition differ by almost 25%. For face recognition, the processing time difference is comparable to the data transfer time.

Since the processing times across applications differ anyway, offloading the same application to distinct servers is analogous to offloading different applications to the same server.

1. As explained in Section 6.5.1, we cannot train in real time, so the applications are fed with synthetic video feeds. The input data for plate recognition and OCR are different, hence the transfer times differ.

Therefore, we implicitly capture the offloading complexity mentioned above by studying a suite of applications.

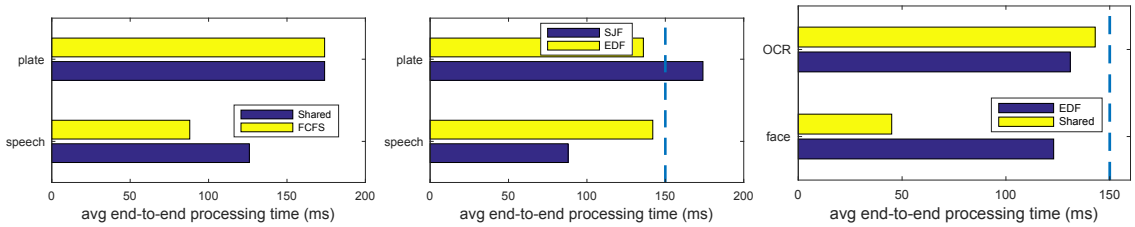
One issue remains, however. The benchmark applications are representative of the continuous applications described in Section 6.2, for which there is an implicit deadline requirement for processing to ensure smooth user interaction. Meeting deadlines can be more important than optimizing for the end-to-end processing time, though the latter is an essential contributing factor to the former. We next explore suitable scheduling metrics to capture desirable performance goals.

6.3.3 Scheduling metrics and algorithms

Recall that our main goal is to schedule data transfer and hence order the offloading requests. Scheduling is a well-studied topic. Common metrics and criteria include fairness, minimizing job time (in our case, this means the shortest network transfer time, or minimal network queuing delay), and deadline-awareness for real-time jobs. We consider these metrics and the associated standard algorithms in turn. Note that there are multiple definitions for fairness. Since we are concerned with a wireless link, we use the common notion of slot-based fairness, which is also consistent with round-robin in the OS.

Fair sharing vs serial execution. Perhaps one of the simplest scheduling approaches is to serve all jobs in parallel, i.e., serving the data transfers from all offloading jobs simultaneously using technique like parallel TCP [279]. Each application can get the same share of the bandwidth, which is referred to as fair sharing. Instead, we argue for sequential offloading (as adopted by First-Come-First-Serve, or FCFS) instead of fair sharing.

Under blocking-based offloading scenarios, a remote execution process cannot make progress until it has received all the data. Consider a scenario where the concurrent offloading tasks are from speech recognition and plate recognition, the former arriving first. Figure 6.3a compares the end-to-end processing time for each task (the sum of the network transfer time and the computation time) between fair sharing and FCFS, the latter clearly outperforming the former. This is mainly because fair sharing incurs higher queuing delay for each task and hence a much higher average network transfer time of 84 ms, 30% more than that of FCFS. This in fact reconfirms that fairness often does not align with optimal



(a) Fair sharing vs FCFS (b) SJF vs EDF (c) EDF vs Sharing

Figure 6.3: The impact of scheduling algorithms on the end-to-end processing time performance [280].

Therefore, *serial execution can achieve better average application processing times by reducing the average network transfer times.*

To pre-empt or not to. The next question is whether to pre-empt any jobs. Devices are not limited to relatively powerful mobile phones, and could be embedded devices with limited memory and high context-switching overhead. For these situations, the cost of moving state between memory and storage to pre-empt one process and bring in the next could be prohibitive. Therefore, *we focus on non-pre-emptive scheduling techniques that can be applied across a range of devices.*

Processing throughput vs deadline. To optimize for performance, perhaps the most intuitive approaches are minimizing the job completion time and meeting job deadlines (in our case, bounding the *end-to-end processing time*). Correspondingly, the canonical scheduling algorithms are Shortest Job First (SJF) and Earliest Deadline First (EDF).

In particular, it might appear natural to minimize the end-to-end processing time, since this could be directly achieved with SJF and *appear* to lead to meeting deadlines. However, this approach can unintentionally penalize one offloading job as a result of *another job* using a less powerful backend. Therefore, the SJF decision should be based on minimizing the network transfer time alone. For example, if we offload OCR and plate recognition concurrently, according to the processing time breakdown in Figure 6.1, the scheduler based on the shortest *end-to-end processing time* will schedule plate recognition first, and produce 179 ms average end-to-end processing time, while that of the shortest *network transfer time* will choose the reverse order of offloading, which will produce 160 ms end-to-end processing time.

Consider the scenario of concurrent speech recognition and plate recognition jobs again.

Figure 6.3b shows that, regarding the average end-to-end processing time, SJF outperforms EDF by only 6%. However, if both had a 150 ms processing deadline requirement, both meet the deadlines under EDF, while only plate recognition manages so when using SJF.

Therefore, *the scheduling decision should primarily depend on the network transfer time instead of the end-to-end processing time; further, if the main scheduling concern is deadline-awareness, minimizing the average processing time with SJF does not work well even in the simplest two-app scenarios.* This is despite the fact that a low end-to-end processing time is essential to meet the deadline.

The tale of the tail. So far, EDF appears to be a winner. However, one consequence of its non-pre-emptive nature is tail latency. EDF is not tail-robust [281], i.e., its performance will suffer when the network transfer times of different applications follow a heavy-tail distribution.

Now consider a face recognition job and an OCR job in the queue. Figure 6.3c shows that both meet their deadlines (150 ms, marked by the vertical dashed line) under both fair sharing and EDF. However, the average end-to-end performance of face recognition suffers under EDF. This is mainly because the network transfer time of OCR is disportionately large compared to that for face recognition. Although OCR should be prioritized given the EDF policy, it appears that fair sharing is the right approach instead. Lack of tail-robustness manifests when the heavy workload essentially blocks the light workload due to non-preemptive scheduling.

Therefore, *EDF suffers when the network transfer time follows a heavy-tail distribution, but sharing can mitigate this problem.*

Summary. To conclude this study, the winning scheduling objective appears to be prioritizing meeting deadlines but also trying to be tail-robust at the same time.

6.3.4 EDF with Limited Sharing

To balance the requirements of meeting application deadlines and avoiding a long tail-latency for network transfer, we augment Earliest Deadline First with Limited Sharing (EDF-LS). The idea is very simple. EDF-LS starts with EDF as the baseline while dynamically determining whether to multiplex the shared link between several transfers. To

simplify further, we only consider two consecutive transfer tasks.

Suppose the two consecutive tasks in the queue, T_1 and T_2, are already ordered based on EDF. Their estimated network transfer times are N_1 and N_2 respectively. The limited sharing will be enabled if and only if the following condition holds: $N_1 > S \cdot N_2$. S captures how much the heavy-transfer task is affecting the other, hence whether sharing is waranted. We set $S = 5$ empirically (Section 6.5.4).

This way, our scheduling algorithm can perform like EDF when the network transfer distribution is short-tailed. When a large transfer task arrives, however, the sharing enabled can allow small tasks to make progress as well. Note that sharing carries a cost, and therefore limiting sharing is essential to ensure the overall performance.

6.3.5 Discussion

The choice of offloading destination. Although the above scheduling algorithm is set in the specific context of current mobile offloading practice, many issues to consider are generic to multiple concurrent offloading sessions. The destinations can be a nearby (edge) server, the cloud, or even another nearby mobile device. For example, we can envisage offloading from a phone to a pad, such as offloading graphics rendering for multi-party gaming. There can also be more complicated link sharing mechanisms. The above expressions can be adjusted to reflect more general offloading scenarios.

Intra-application dependencies. So far we have assumed that the individual modules within an application are independent from one another. It is conceivable that two modules to be offloaded may instead be logically correlated. For example, one module may need to wait for the result from another module. We believe such correlation should be managed by the application itself by setting appropriate per-module deadlines to reflect the correlation, and therefore we do not consider such dependencies in this paper.

Fending off greedy applications. Another implicit assumption in our discussion is that each application will suggest reasonable deadlines for their offloading requests. If instead, a greedy (or malicious) application wants to game the system, it can intentionally set a tight deadline to gain more network resource. One solution to counter this is to track the variance of the end-to-end processing time distribution of each application and infer whether any

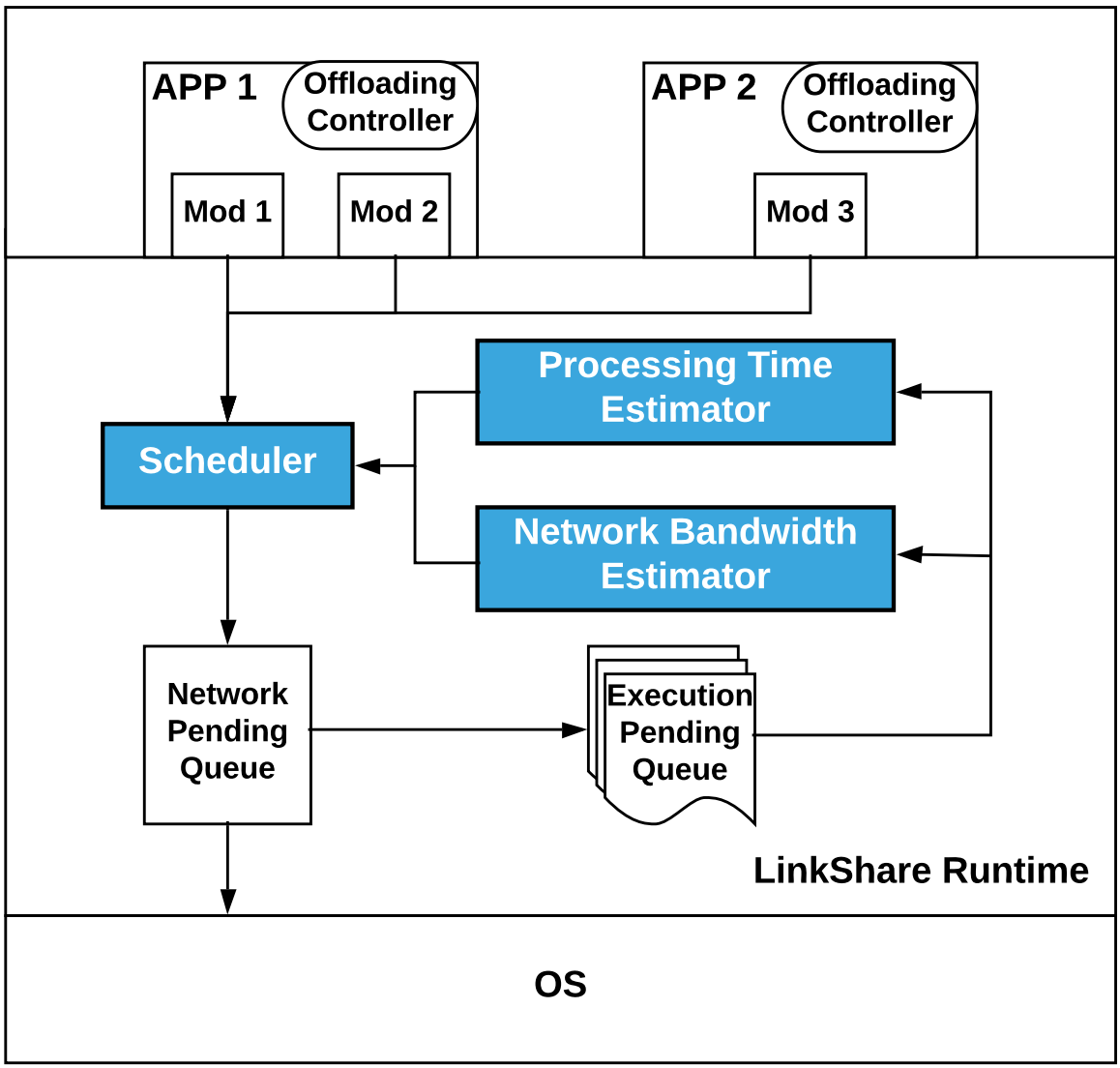


Figure 6.4: LinkShare architecture.

application had been favored throughout.

6.4 LinkShare as a service

We next build a system level service, LinkShare, that coordinates multiple offloading jobs. Fundamentally, LinkShare is an extension to the operating system scheduler. It takes into account the wireless link bandwidth, remote processing capability, application processing deadline requirements and their effects on the actual end-to-end processing times. To support the central goal of making scheduling decisions, it collects information of past runs, estimates future execution times, and computes an appropriate schedule.

LinkShare sits between the applications and the operating system. Figure 6.4 shows the system architecture. There are two control paths: *application driven scheduling* and *background monitoring*. We describe the individual modules next.

Note that our system architecture is independent of the exact scheduling algorithms. Different scheduling algorithms can be substituted if needed.

6.4.1 Application driven scheduling

The application first decides whether to offload any workload remotely using an existing offloading runtime or its custom decision module, as discussed in Section 6.3.1. If offloading is needed, it sends the request to LinkShare, along with the deadline for completing the remote processing (e.g., in terms of the longest permitted processing time). LinkShare then determines the order to execute the offloading requests, in terms of the order of data transfer to the offloading destinations, using the Earliest Deadline First with Limited Sharing (EDF-LS) detailed in Section 6.3.4.

All offloading requests from applications are first enqueued in the *network pending queue*. Given our EDF-LS algorithm, this is essentially a priority queue, where the network transfer completion deadline (computed from the time instant at which LinkShare receives the request from the application, the estimated remote processing time and the processing deadline specified by the application) serves as the priority indicator.

The scheduler continues to empty the *network pending queue* as long as there are pending offloading requests. Once network transfer is completed, an offloading request is moved from the *network pending queue* to the corresponding per-application[2] *execution pending queue* and remains there until a response is received from the remote server.

6.4.2 Lightweight background monitoring

In the background, LinkShare tracks the states of each scheduled request, including the size of data transfer s_d, and the start and end time of the network transfer t_s and t_e.

Remote processing time estimation. To assess whether an application deadline can

2. Strictly speaking, "per-application" refers to "per-module" in this section.

be met, it is essential to estimate the remote processing time. The main challenge here is to the intrinsic variability of processing times. Take popular learning-base mobile applications as an example. The variability may arise from (a) caching, (b) using multiple models simultaneously to speed up inference, (c) input variation, and (d) other customized optimizations [148].

Given all the variability, we estimate the average processing time (i.e., per-application) instead of the per-job processing time (i.e., for each data transfer). This is for two reasons.

First, it simplifies the system implementation significantly to be lightweight on a mobile device. In contrast, current mainstream per-data-transfer processing time estimation techniques are all learning based and incur high computation complexity. [13]

Second, since the mobile applications of interest to us usually have soft real-time requirements, an accurate estimate of the *average* processing time is sufficient for our scheduling goal of meeting deadlines.

We adopt the exponentially weighted moving average (EWMA) to estimate the average, per-application remote processing time. The estimate T_{est} is initialized to 0 and updated in an event-driven manner. The completion of each offloading request at time t_r triggers an update for T_{est} based on the previous estimate and the observed processing time ($T_{cur} = t_r - t_e$) incurred by this completed request. Empirically, we set the smoothing parameter λ to 0.3.

$$T_{est} = T_{prev-est} \cdot (1 - \lambda) + \lambda \cdot T_{cur}$$

Network bandwidth estimation. The sharing component of EDF-LS is triggered based on the projected network transfer times. Therefore, network bandwidth estimation is also essential to LinkShare. Since wireless link bandwidth is known to fluctuate, we again resort to the classic EWMA. Triggered when an offloading request completes its network transfer, the bandwidth estimate is updated as follows:

$$B(t + T) = B(t) + \alpha \cdot (B(t) - \frac{u(T)}{T})$$

Where $B(t)$ is the estimated network bandwidth at time t, T is the interval between

two updates, u is the number of bytes sent over T, and α is the smoothing parameter. This way, we can estimate the network bandwidth without incurring any network overhead.

While these above estimation algorithms work well in our evaluations, we realize that they represent a set of very simple heuristics. More sophistication can be taken into consideration to make these estimation more robust and accurate. For example, when estimating remote processing time, the estimation accuracy will increase if we can coordinate the real-time workload information on remote servers. Similar argument can be applied to network bandwidth estimation. We defer these and other improvements to future work, and currently choose to focus on the potential benefits brought by scheduling of multiple offloading applications.

Security considerations. Tracking the variance of the end-to-end processing time distribution of each application might not be resistant to malicious applications that overclaim deadlines from launch time. Fortunately, proper access control can efficiently protect our system from these applicaitons. This can be achieved by incorporating a reputation system (such as Credence [282]) into LinkShare. Each offloading request can be tagged with its application source. The tracking of end-to-end processing time of each offloading request can help to establish a reputation record for each application. The application whose deadline is always far beyond its end-to-end processing time can be identified and barred from time to time.

6.4.3 Implementation details

We implement LinkShare as a background application level service in Android Nougat OS with API version 24.

Since our scheduler runs in the user space, we enforce the schedule computed by setting the priority of thread that processes the scheduled request to `Thread.MAX_PRIORITY`, using the Andriod API `Thread.setPriority()`.

LinkShare APIs. LinkShare exposes the following APIs to the applications: `offloadRequest`, `decisionMade`, `notifySent`, and `notifyComplete`.

`offloadRequest` is used by an application to send an offloading request to the scheduler, containing the module name, input data size, process id, timestamp and the end-to-end

```
// Conventional application code
if(offloadMode == true) {
     asyncRemoteRecognition.sent(image);
     while(!asyncRemoteRecognition.complete()) {
       continue;
     }
     result = asyncRemoteRecognition.result;
}

// With the scheduler service
offloadRequest(modName, timeStamp, pid, inputDataSize, deadline);

while(true) {
     if(receive decisionMade) {
          offloadMode = decision.offloadMode;
     }
}

if(offloadMode == true) {
     asyncRemoteRecognition.sent(image);
     notifySent(modName, timeStamp);
     while(!asyncRemoteRecognition.complete()) {
       continue;
     }
     result = asyncRemoteRecognition.result;
     notifyComplete(modName, timeStamp);
}
```

processing time deadline. Once a schedule has been computed, the scheduler sends a `decisionMade` message to all applications with pending requests, indicating the module to be offloaded first.

`notifySent` is used by an application after completing sending its data to the remote server. This is intended to notify the scheduler to schedule the next offloading request in the queue and trigger the network bandwidth estimator to update the runtime bandwidth `notifyComplete` is used by an application when a computation job is completed. This notifies the scheduler to mark this scheduled request as completely and trigger the remote processing time estimator to update the estimated remote processing time for this specific application.

The following pseudocode shows code snippets for a face recognition module to send an offloading job with and without our scheduler service. The lines in bold indicate the code patch needed to use our service. All changes are on the mobile side, with little overhead.

Application code change. We opt to expose APIs to applications instead of making

LinkShare transparent for two reasons. First, the application needs to set the deadline requirement explicitly anyway, since this cannot be inferred accurately. The system can only track the processing times of past runs and estimate the *bounds* (the shorest or the longest) on the deadlines that can be met. Second, the scheduling framework is not restricted to computation offloading. Therefore, we highlight the APIs needed to coordinate multiple concurrent network-bound applications in general.

Offloading framework for experiments. We also implement an offloading framework for experiments. This is independent from our scheduler service.

We define our own communication interface for service-application communication using Android Interface Definition Language (AIDL [283]). AIDL makes it easy to handle multi-threaded asynchronous inter-process communication.

The communication framework between our mobile device and server is built on gRPC [284]. gRPC is portable and provides a stream model, which we use to implement an asynchronous communication channel between a client and the server. The client does not need to wait for server response before sending another request.

6.5 Evaluation

6.5.1 General setup

Application scenario, benchmarks, and test data. We emulate a Gabriel [31] like scenario where multiple recognition applications concurrently operate on the camera and audio feeds from a phone and collectively provide assistance to the phone user to interact with the environment. Each video frame is processed simultaneously by one or more applications. We built four benchmark applications: face recognition, plate recognition, speech recognition, and an optical character recognition (OCR) application. These mimic similar commercial applications but are simplified to contain only the most relevant library functions. They are also extensively instrumented to provide the measurements we need, which we cannot obtain from commercial applications.

The *Face recognition* module recognizes the faces in a given video frame. We build the module using the Local Binary Patterns Histograms (LBPH) face recognizer in OpenCV [285].

The experiment data are taken from the unconstrained facial images database [286], which is a set of real-world photographs selected from the large photobank [287]. We use the cropped images dataset, where faces were automatically extracted from the original photographs. This set contains images of 605 individuals, on average 7.1 images per person.

The *Plate recognition* module recognizes the car license plate in real time. We build this using the OpenALPR library which is a popular open source plate recognition library [288]. We use the plate dataset from [289], which contains 500 images of the rear views of various vehicles, the resolution of each image is 640×480.

The *Speech recognition* module is built on the CMU Sphinx speech recognizer [290]. This application can recognize a set of spoken commands given by the users, including "open application", "call home", "close application", "take a picture", "tell me the location" and "tell me the time". The corresponding input data are the voice recordings from different occupants in our lab, 60 samples in total, 10 for each command. Note that our speech recognition application is completely processed on a remote server, like the main-stream speech driven assistant *Siri*.

The *OCR application* is built on the tesseract open source OCR engine [291]. It extracts text information from images. This is to emulate a scenario where users go abroad and do not understand the local language, so they use their phones to extract the textual information from signposts and translate it to familiar languages. We use the KAIST Scene Text Database [292] as the input, which contains signposts and brands under different light conditions.

Note that these applications all contain specific pre-trained models. During the run time, they only perform *inference* on any input data. This is because we cannot train these applications in real time yet. Therefore, we emulate a video feed using data from the test sets specified above to generate 4 parallel feeds[3]. For each application, we generate a long sequence by randomly drawing data from the corresponding test set and feed this sequence to the application. Across applications, we make the corresponding frames similar in size

3. While this approach may not fully mimic the inter-frame correlation in a real video feed, it does not affect our current evaluation. In a real video feed, we could leverage temporal correlation between frames and skip a few runs for some applications as a further optimization, but this is orthogonal to making scheduling decisions.

as much as possible.

Canonical testbed setup. We use a Samsung S9 smartphone as the mobile device, with an octa-core (2.7 GHz quad-core M2 Mongoose + 1.7 GHz quad-core Cortex A53) processor, 4 GB of RAM and 64 GB of internal storage. We use two servers, each with an Intel core i5-4570 processor (Quad-core, 6 MB Cache, 3.2 GHz) and 8 GB 1600 MHz DDR3 Memory.

Unless otherwise stated, the network upload speed is 10 Mbps. We use this as the baseline since the average wireless bandwidth measured was 8.63 Mbps in a recent network report [293]. This is the perceived TCP transfer throughput, not the physical layer rate on the wireless link.

The network transfer is over the WiFi link from the phone to the nearest access point. Since the WiFi networks in our lab are usually faster than the reported average speed and fluctuate between successive runs, we emulate a stable link at a target upload speed by rate limiting the WiFi transfers. We consider other network conditions later.

We also explored using other phones and server capabilities (both the network latency to the server and the processing capability). We find these do not change the qualitative observations. As long as the server is much more powerful than the phone such that offloading is worthwhile, the performance of our scheduler is indifferent to the phone processing capability. Further, the effects of server conditions are captured in applications showing different combinations of the network transfer and server processing times. Therefore, we only report results based on the canonical setup.

Note that our application benchmarks are not fully optimized, so local processing might take even longer than it could be. To offset that, we also use less powerful servers. This means that the effect of network transfer time overhead in our setting is in fact less pronounced than in scenarios with faster phones and servers. We will discuss this in the context of upload speed scaling in Section 6.5.3.

6.5.2 Scheduler Performance

We aim to answer two questions: (a) is the additional scheduling added by LinkShare essential? (b) How well does EDF-LS perform in terms of meeting application deadlines? Both can be addressed together by comparing the performance of several scheduling algorithms.

Scheduling algorithms for comparison. We compare the following algorithms: (a) *Fair sharing*: This algorithm achieves slot-based fairness on the wireless link. (b) *First come first serve (FCFS)*: this is the most naive non-preemptive scheduling algorithm, as well as the default network scheduler; therefore it is a proxy for the scheduling decision in the absence of LinkShare. (c) *Shortest job first (SJF)*: this is the algorithm that minimizes the network transfer time. (d) *Earliest deadline first (EDF)*: this is the optimal deadline-aware algorithm among the non-idling non-preemptive scheduling algorithms. (e) *Earliest deadline first with limited sharing (EDF-LS)*: our algorithm.

Workloads. Given the network transfer time distribution of individual application benchmarks, we assemble two workloads, referred to as *heavy-load* and *light-load*.

The heavy-load includes all four benchmark applications running concurrently. This is a general case, and likely more common in future. The light-load setting involves only face, plate, and speech recognition offloading concurrently, without OCR. This division is motivated by the network transfer time disparity. On average, the network transfer time of OCR is nearly the sum of the transfer times for the other three applications.

Setting the processing deadline. Since the actual deadline requirements of the commerical applications are proprietary, we determine the deadline based on the tail of the processing time for each application. We consider three options, using the 90th, 95th and 99th percentile of all these applications to determine their individual deadlines respectively.

Note that we assume the application deadline and the request sending interval are coupled together. This is reasonable since these applications operate on continuous video feeds, and the frame processing deadline partly arises from the need to keep with the frame capture rate. Unless stated otherwise, therefore, we will set the sending interval between two consecutive data frames in each application to be their individual deadline.

Performance metric. We use the deadline miss rate to assess deadline-awareness. This is defined as the ratio of the number of data frames that miss the deadline to the total number of data frames sent, turned into a percentage.

Light load. Here we set the deadline as 1.2× the 90th, 95th and 99th percentile of the end-to-end processing time. Figure 6.5 shows the performance of different scheduling algorithms performance given these deadline settings. We first note that each application favors specific

scheduling strategies, and one often benefits at the expense of another.

FCFS vs EDF-LS: As we can see from all three figures, FCFS suffers when considering face recognition (the least bandwidth hungry in this case) but outperforms all the other scheduling algorithms when considering plate recognition (the most bandwidth hungry in this case). This is because the offloading request arrival pattern determines the performance of FCFS. As the request interval of face recognition is the smallest among these three applications, the probability of another application blocking the next data frame transfer is the highest for face recognition among these applications.

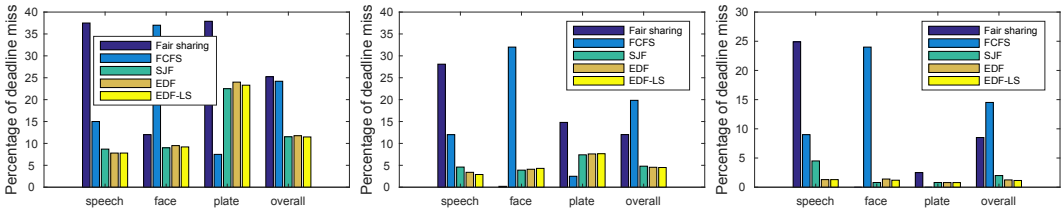
Recall that FCFS is a proxy for not employing LinkShare at all, i.e., managing offloading requests without adjusting the network transfer order. Figure 6.5 clearly shows this additional scheduling is essential.

Fair sharing vs EDF-LS: In contrast, fair sharing outperforms all the other scheduling algorithms for face recognition, but performs the worst for the other two applications. This is because, intrinsically, sharing is beneficial to the application that is less bandwidth hungry. However, it is at the cost of performance degradation of other applications that are more bandwidth hungry.

SJF vs EDF-LS: When given tighter deadlines, SJF shares similar performance to that of EDF-LS. When the deadline requirements are relaxed to 1.2× the 99th percentile, EDF-LS outperforms SJF for speech recognition application and delivers similar performance to the other applications. The main reason is SJF is that not aware of different deadline requirements across applications. Speech recognition incurs a slightly shorter network transfer *time* than plate recognition on average. However, in terms of the network transfer *deadline*, plate recognition has a much longer deadline than that of speech recognition. This means transfer the data frame for speech recognition before plate recognition is a better choice in general, but making scheduling decisions based on the network transfer *time* is a bad idea.

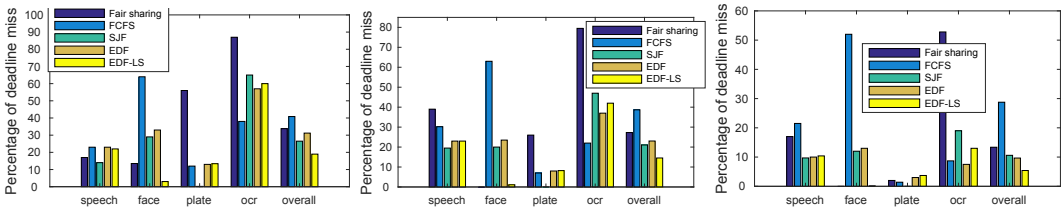
EDF vs EDF-LS: In the light workload scenario, EDF and EDF-LS share similar performance across all applications. This is expected, as light load rarely triggers sharing, and the performance of EDF-LS should converge to that of EDF. Sharing may incur a penalty in this case, as it slows down one job without helping another job much.

Heavy load. We set the individual application deadline to be 1.5× the 90th, 95th and 99th



(a) 90th (b) 95th (c) 99th

Figure 6.5: Deadline miss rate under light load at 10 Mbps



(a) 90th (b) 95th (c) 99th

Figure 6.6: Deadline miss rate under heavy load at 10 Mbps

percentile of the end-to-end processing time of the respective application. We experimented with factors ranging from 1.2 to 1.5, and found that 1.5 is a suitable value to allow all these applications to mostly meet their deadlines.

Figure 6.6 shows the performance of different scheduling algorithms given three different deadline settings. We mainly discuss the comparison between EDF and EDF-LS in this case, as the performance of the other scheduling algorithms is similar to the situation under light load.

The main difference between EDF and EDF-LS can be seen in the performance of face recognition and OCR. EDF-LS largely outperforms EDF for face recognition, at the cost of slight performance degradation of OCR. Specifically, the performance improvement margin is 30% given a tighter deadline (the 90th percentile case) and still 15% given a loose deadline (the 99th percentile case). This highlights the importance of mitigating head-of-queue blocking that would have originally happened to EDF. In fact, this blocking issue is not restricted to EDF. It also plagues other non-preemptive scheduling algorithms like SJF and FCFS. By mitigating this with limited sharing, we can reduce the missing rate of face recognition to less than 3%.

Summary. The additional scheduling added by LinkShare over the default OS scheduler is essential, and EDF-LS can achieve up to 30% reduction in the deadline miss rate compared

to the baseline EDF.

6.5.3 Impact of Network Conditions

As network transfer is the major bottleneck in our system, intuitively the link quality dictates the performance of EDF-LS. We consider various bandwidths on steady links and two real traces of fluctuating networking conditions.

Steady links. Since wireless networks continue to evolve, we may expect increasingly higher bandwidth. We are therefore interested to see whether the problem remains under better network conditions.

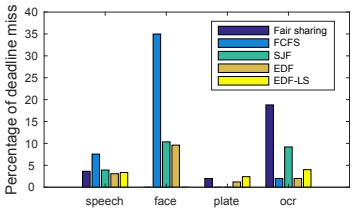
We repeat the previous heavy-workload experiments under two additional bandwidth settings, 15 Mbps and 20 Mbps, as a proxy to show the system performance as the upload bandwidth increases. Note that, as the network bandwidth will affect the estimate of the end-to-end processing time, the deadline under a higher network bandwidth will be tighter than that under 10 Mbps.

Figures 6.7a and 6.7b show the performance with 15 Mbps and 20 Mbps upload speeds. These show that EDF-LS consistently incurs the lowest miss rates for face recognition and similar miss rate compared to EDF for other applications.

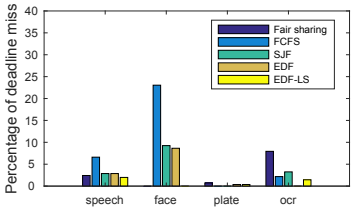
We see that the increasing network bandwidth has the largest impact on the performance on fair sharing and FCFS. This is because higher bandwidths reduce the network queuing time and thus mitigate the effect of bad scheduling decisions.

One interesting observation is that, for face recognition, the performance of EDF and SJF hardly change even as the network bandwidth goes up. The reason is that deadline misses occur due to the blocking nature of these two algorithms (face recognition is blocked from data transfer by a network-bound job), not its own network transfer time. The blocking effect remains even at higher network speeds.

More generally, even as the bandwidth increases further, the scheduling problem is likely to remain, because the server processing capacity is likely to increase in tandem. Fundamentally, the cause of missing deadlines in our context is the relative network queuing delay, i.e., the absolute queuing delay divided by the end-to-end processing time. This depends heavily on the ratio between the network transfer time and the remote processing

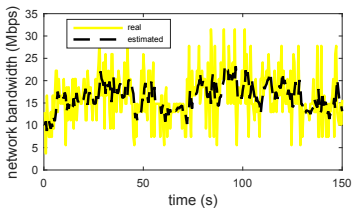


(a) 15 Mbps

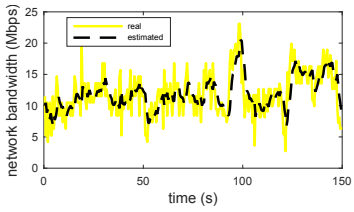


(b) 20 Mbps

Figure 6.7: Performance under different network upload speeds

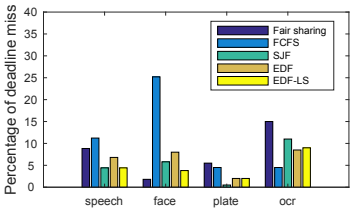


(a) WiFi

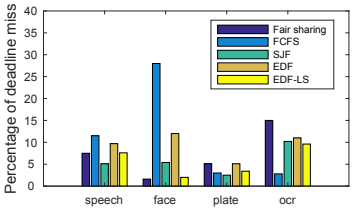


(b) LTE

Figure 6.8: Real WiFi and LTE traces



(a) WiFi



(b) LTE

Figure 6.9: Performance under real WiFi and LTE network conditions

time.

Fluctuating network conditions. Being aware of the potential influence of mobile network conditions to the evaluation results, we next study the performance as the link condition fluctuates.

We first measure WiFi and LTE bandwidths by running `iperf3`, using our mobile phone as the client sending traffic to one of our servers. Again, these are the TCP throughput numbers, instead of the raw physical layer rates. The WiFi trace is captured in a cafe near campus, while the LTE trace is collected as one walks from the student dormitory to the department building. We record the bandwidth every 0.5 second, which is the minimal interval supported by `iperf`. Each trace covers a 150-second period.

Figures 6.8a and 6.8b plot the bandwidth timeseries for the WiFi and LTE traces captured respectively (the "real" lines). The "estimated" lines in these two figures also show the estimated network bandwidth using the bandwidth estimation module in LinkShare. Our module indeed manages to track the trend of average network bandwidth changes.

We then replay these traces to evaluate the performance under fluctuating network conditions. We assume the same deadline and request sending interval settings (1.5 times the 99th percentile of the end-to-end processing time) as that in the 10 Mbps, heavy-workload for both WiFi and LTE.

Figures 6.9a and 6.9b show the performance under WiFi and LTE respectively. EDF-LS

can still show similar performance to that under steady links.

An interesting observation is that EDF-LS outperforms EDF for *all* applications, not just face recognition. This is especially pronounced for LTE. Similarly, fair sharing fares much better over fluctuating links than steady links. The reason for both observations is similar to what happened at increasing network upload speeds. Sharing can intrinsically benefit from a larger bandwidth. In a highly fluctuating network, the network bandwidth spikes can be better used when multiplexed between different applications.

6.5.4 Microbenchmarks

Finally, we performance several microbenchmarks to assess the design and overhead of our scheduling algorithm.

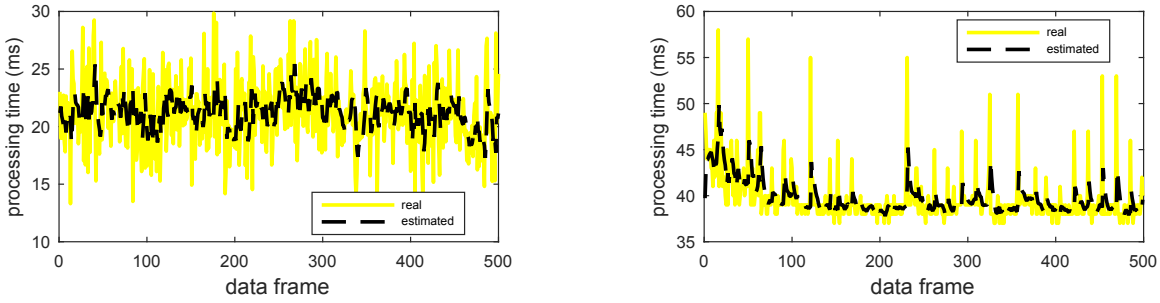
Scheduler overhead. The scheduler overhead is mainly from the scheduling decision time. This takes less than 2 ms.

Processing time estimation. We already assessed the accuracy of the *network bandwidth estimation* module in the previous subsection. Here we use face and speech recognition as two examples to assess the accuracy of the *processing time estimation* module, shown in Figure 6.10.

Face recognition is a representative application with relatively stable processing time pattern, while speech recognition represents a version with more fluctuation. OCR and plate recognition share similar processing time patterns as that of speech recognition. We run each application 500 times and randomly selected input data from their respective dataset. As we can see from the figure, our processing time estimation module manages to track the trend of the average processing time changes. Quantitatively, the root mean square errors of the processing time for speech and face recognition are 2.56 ms and 1.54 ms respectively.

Sharing parameters. Recall that the sharing component of EDF-LS is triggered based on a threshold, and the parameter S hints at the disparity between the projected network transfer times between two consecutive offloading requests (Section 6.3.4). Therefore, we need a suitable value for S.

To avoid the randomness due to the input stream, we first run the heavy-load combi-



(a) Face recognition (b) Speech recognition

Figure 6.10: Processing time estimation

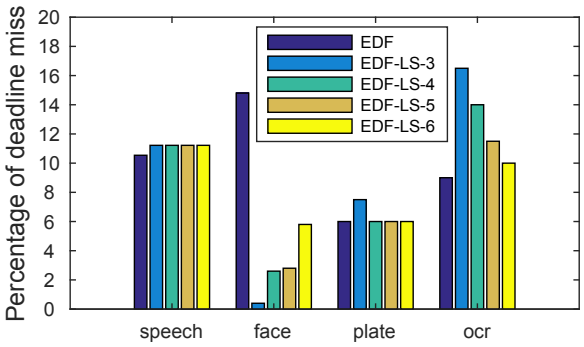


Figure 6.11: The value of sharing parameter

nation under the 99th percentile deadline constraints with $S = 3$ and 10 Mbps network upload bandwidth. We record the processing time and file size for each data frame. Then, we replay the trace in simulation and try different S values to assess its impact.

Figure 6.11 shows the results for $S = 3$, 4, 5, and 6. The main difference comes from the performance of face recognition and OCR. The higher the S value, the less benefit face recognition gained from the limited sharing. A good trade-off between respecting the deadline and mitigating heading-of-queue blocking is achieved when the S value is around 4 to 5. As S increases (i.e., requiring higher transfer time disparity before enabling sharing), the performance of EDF-LS approaches that of EDF, which means limited sharing is rarely activated. Based on this analysis, we set the sharing parameter value to be 5.

Discussion. Another dimension of our algorithm is the choice of the maximum number of workloads that can share the link bandwidth. The optimal number is workload dependent. For our workloads, sharing between two consecutive jobs appear to be optimal and the most robust choice.

Another point worth mentioning is that, for now, the choice of sharing parameter S aims

at minimizing the average percentage of deadline missing across all workloads. That is to say, implicitly, we treat all the workload applications the same. However, the preference of different user might make the utilities of these workload applications different. We believe it is an interesting future work if introducing utility of each application as a factor when making scheduling decision.

6.6 Related work

For most of the last decade, the computation-intensive requirement from mobile applications has been met with computation offloading to a cloud server. New dedicated hardware has also emerged in recent years. However, existing solutions are limited as new applications are on the horizon, calling for more inter-module or inter-application coordination. We will discuss relevant resource management approaches in the context of common offloading practice.

Dedicated Hardware. The Google Tango project [30] uses dedicated hardware to power computation-intensive workloads locally. It does not depend on external resources and simplifies application development. Resource management is simply delegated to the operating system. However, the need for additional hardware and often high power consumption are ill-suited for wearable devices like Google Glass, which generally require remote support.

Single-module Offloading. Computation offload to the cloud has been explored extensively in the last 15 years [63–70]. Several recent works [12–16, 71] focus on seamlessly partitioning the workload spanning the mobile device and the remote server.

They assume only one foreground application at a time. All other applications within the same device are treated as background processes, with lower priorities. This is indeed true for previous scenarios. However, emerging applications are more sophisticated and all the modules concerned must be served simultaneously, given the same priority. Single-module offloading techniques are insufficient.

Multi-module Offloading. MCDNN [72] provides a common platform for multiple applications concurrently utilizing deep neural networks (DNN). When reasoning about the workload split between the device and the cloud for each DNN application, MCDNN does

coordinate on-device resource usage between applications. However, the coordination is specific to DNN applications run on MCDNN and integrates decisions to trade off classification accuracy for less resource usage.

Cloud-centric offloading. Gabriel [31], mentioned earlier, offloads all its computation intensive workloads to the nearby cloudlet, where both the control and computation units are located. The underlying assumption is all of these applications can be processed on a single backend server. However, even for Google Glass, apart from a handful of built-in applications like Google Now and Google Maps, most applications are from third-party developers, who may maintain their own servers. It may be impractical to make all these applications offload to the same destination. In that case, the cloud-centric offloading scheme can not be applied. Instead, a device-centric approach seems more promising.

Deadline-aware scheduling. Deadline-aware scheduling has been studied extensively, most recently in data center networking [280, 294, 295]. [280] controls the network transfer rate to meet the deadline, but rate control is difficult for mobile scenarios. [294] pre-empts flows to approximate a range of scheduling algorithms. However, we explain earlier that the overhead of link switching in wireless is high.

Summary. Although there is a multitude of offloading schemes to handle computation-intensive workloads on mobile devices, none appears to provide the coordination needed between multiple offloading modules to serve the emerging applications outlined above. LinkShare fills in this gap with a generic on-device scheduling service for all applications interacting with backend servers. LinkShare augments standard deadline-aware scheduling to suit our needs.

6.7 Summary

In this chapter, we investigated concurrent and continuous mobile-cloud interactions in the form of simultaneous offloading jobs. These share the wireless link from the mobile device but might involve different server backends. Therefore, we need device-centric management, instead of the more common cloud-centric control.

We build a system-level service, LinkShare, that wraps over the operating system sched-

uler to coordinate among multiple offloading requests. We study the scheduling requirements and suitable metrics, and find that the most intuitive approaches of minimizing the end-to-end processing time and earliest deadline first scheduling do not work well. Instead, LinkShare incorporates limited sharing between consecutive workloads in the case of heavy-tailed distribution. LinkShare is implemented in Android. Extensive evaluation shows that adopting this additional scheduler is essential and that EDF-LS can achieve up to 30% reduction in the deadline miss rate compared to the baseline EDF.

The issues we studied are not specific only to computation offloading to a remote cloud. Even with the advent of edge computing, the same bottleneck remains. LinkShare as a framework is also applicable when the offloading destination is a nearby server or another IoT device instead. We believe this is a first step towards coordinating an IoT ecosystem tethered to the mobile device from the perspective of the mobile frontend.

Chapter 7

Discussion: Edge Infrastructure Provisioning

Computation offloading is a standard solution for latency-sensitive and computation-heavy apps, and there are huge amount of works talking about offloading app codes to the cloud. However, in order to further reduce the latency, mobile edge computing has received increasing attention nowadays, and there are multiple proposals for edge computing infrastructure such as smart access point, workstations in base stations, cloudlets, etc. Despite that, there is a lack of proper business model about how to organize these heterogeneous and decentralized edge infrastructures and provide clean and easy-to-use interfaces to app vendors. In this chapter, we study the requirements for providing the edge service, and argue for sharing economy model to organize and provide the service. We further study the challenges and possible solutions for implementing such sharing economy model.

7.1 Introduction

With the emerging of edge computing, people can deliver even lower latency for applications by placing computation resource close-by. However, it is extremely difficult to figure out the proper edge service to use. Traditionally a third-party developer will have to choose the right cloud provider [296] and server setup (including the number of VMs to balance demands from different geographical locations and over time), and maintain the server instances.

Cost issues aside, the technical challenge of identifying the correct server configurations (i.e., VM specifications like CPU cores, network bandwidth and storage) to meet all demands can be non-trivial. In edge computing case, this problem becomes even more severe, since current edge system proposals (cloudlet, smart base stations, smart AP, etc.) have more heterogeneous settings (in terms of hardware type, or even pricing model, QoS guarantee, etc.) without a unified standard.

Therefore, learning from the recent trend of cellular access democratization [297, 298], we propose a sharing economy model for mobile edge computing, which can help organize edge service providers under different control domains, and provide app developers with descent and clean interfaces to use and pay for the edge resources. More specifically, we propose a Uber-like system called Eber to manage the edge service, which has the following functionalities:

1. it allows different edge service providers with heterogeneous hardware settings to join in and jointly provide the edge service to app vendors without the edge service providers to hardcode anything or know each other's existence;

2. it hides the underlying infrastructure complexities and provides clean interfaces to app vendors to offload their app codes and pay for the edge resource they have consumed;

3. it can protect app vendor's sensitive data and avoid curious edge service providers probing user privacy;

4. it can protect edge service provider's confidential information (e.g., hardware type, workload information, etc.) from Eber and other edge service providers;

5. it can automatically figure out the most proper edge server for the incoming offloading requests without the app developers worrying about the details;

To summarize, this chapter makes two contributions: first, we propose a sharing economy model for organizing edge servers under different control domains and providing clean interfaces for app developers; second, we discuss the key design points and their challenges for implementing such sharing economy model.

7.2 Motivation

In this section we talk about why we need a sharing economy model to provide the edge service.

7.2.1 The need for on-demand edge service

Existing mobile offloading work [12–16] implicitly assume a *fixed* offloading destination, typically a server name hard-coded into the application source code or the nearest server as suggested in the emerging edge and fog computing paradigms [60]. The former requires determining the best server setup at build time as the offloading destination, while the latter is ill-suited to handling non-uniform application usage patterns.

The complexity of determining server locations. To statically specify the offloading destinations in the application source code, application developers need to set up server side runtime environment. For those in large companies, this could be deployed in their own data centers, while third-party app developers rent VMs from, say, EC2 [299]. This requires making difficult choices.

As more apps are emerging as latency sensitive [288,300,301] but the users could use the apps anywhere, application developers need to decide the geographic distribution of their servers. Further, We expect this issue to be exacerbated by the emergence of mobile edge computing (MEC) and Fog Computing to reduce latency [61, 302], which means that the *remote* (edge or cloud) server will vary more, affecting the processing performance.

The complexity of resource provisioning. Identifying the server location is only the first step, we also need to specify how many machines are needed, what are the machine types, their cost, etc. Even if we took great pain to identify the offloading destination beforehand, a static destination would incur two problems for resource provisioning due to non-uniform distribution of the application usage pattern spatially and temporally. This is more problematic for edge servers.

First, *applications* exhibit different temporal engagement patterns. [303] studies the application logs from 230 K mobile apps and 600 M unique daily users, and shows that the number of application sessions could vary by 6× throughout the day. Moreover, the usage

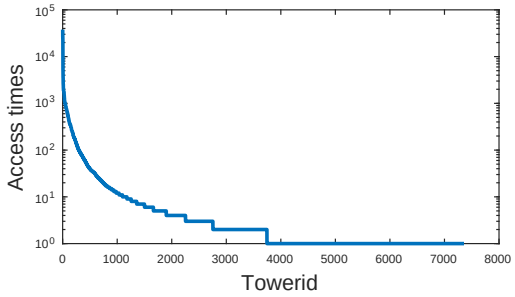


Figure 7.1: Cell tower access patterns

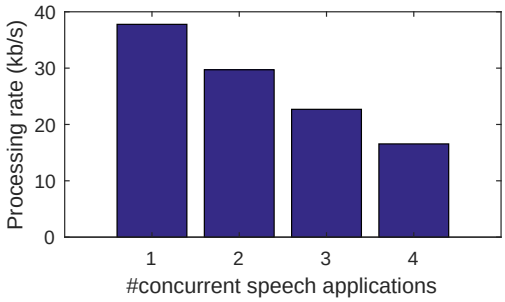


Figure 7.2: Illustration for speech recognition contention

patterns for different application *categories* also vary during the day and across days. For example, educational and finance apps see more active usage from 7am to 4pm on workdays, while sport apps are usually active between 6pm and 10pm at weekends.

Second, *servers* experience different application access patterns spatially. Figure 7.1 shows the tail distribution of the access pattern of cell towers in a published trace [304]. We believe this is would also be indicative of the individual server access pattern if the cell base stations also serve as edge servers. A popular, "hot-spot" server may have difficulty coping with the huge amount of workload. Even if we could provision the computing resource accordingly, the resource utilization will fluctuate throughout the day. There might be a large gap between peak and average utilization, which is inefficient. Conversely, for less popular servers, either the computing resource will likely be under-utilized most of the time or they cannot handle potential peak demands of the day.

The diversity and increasing number of IoT devices [305] will further exacerbate these issues, as the *local* device capability will vary more, hence producing more different offloading jobs.

Both problems can cause load imbalance across different offloading destinations for the app, thus decreasing the overall system resource utilization and application performance. To illustrate the contention effect, we run a simple experiment of multiple speech recognition instances. We first start one instance on the server and measure its processing rate, and then keep adding more speech recognition instances. Figure 7.2 shows that the average processing rate drops from 38 kbps to 17 kbps, which corresponds to a processing time increase from 255 ms to 581 ms. A heavily loaded server nearby may incur a longer end-to-end processing latency than a lightly-loaded server further away.

Solution. Given the myriad of configuration choices and the workload dynamics, it is difficult to determine the best offloading destination at build time. This motivates *on-demand* offloading destination determined at runtime, to find the server nearest to the user that can handle the load. Our solution is *offloading as a service*, so that we can decouple the app development from the server infrastructure.

7.2.2 The need for economic efficiency

Based on previous discussion, it is required to provide on-demand edge service. Due to the low-latency nature of edge computing, the edge servers have to be close to apps, so they need to be placed in cities or towns. There are three options:

1. one entity runs all edge servers in all cities or towns;
2. each entity runs all edge servers in each city or town;
3. multiple entities run edge servers together in each city or town;

From the edge provider's point of view, the economic efficiency is the key concern for providing the edge service. For the first case, it will incur high setup cost for one entity to deploy all edge servers in all cities or towns. What's worse, the edge servers could suffer from hardware/network failures, malicious attack, etc., so the entity has to take care of the fault tolerance, security guarantee, etc. for all edge server clusters, which makes it infeasible for one entity.

For the second case, it is affordable for one entity to only take care of one city or town. However, in this case, the app developers are forced to deal with a huge amount of edge service providers, which may have various pricing models, QoS guarantees and service agreements, etc., so it would be extremely tedious for app developers to choose edge service providers. Therefore, we argue that this is not a good business model for edge service either.

For the third case, this is a sharing economy model where different entities could work together to jointly provide the edge service. In this case, each entity can take care of their own edge server clusters in a distributed manner, thus it is more scalable in terms of fault tolerance, security guarantee, etc. Furthermore, the setup cost is manageable for each

entity: they can decide the number of edge servers as well as the server setup in a flexible way based on their own capability. Besides, currently people have various proposals for how to provide the edge service: cloudlets, basestations with powerful servers, smart AP (access point), etc. These proposals are suitable for certain scenarios: in small cafes, or cellular traffic, or home wireless traffic, etc., and a sharing economy model enables them to work together without requiring one entity to take care of all these scenarios simultaneously, thus it is more economic efficient.

7.2.3 The need for an easy-to-use interface

From the app developer's point of view, they want an edge service which provides an easy-to-use interface which can automatically take care of a bunch of things such as: finding most suitable edge servers, conducting authentication check, providing security check, calculating the amount of work each edge server has done, etc. This interface needs to hide the underlying infrastructure complexities and complicated internal business and technical details.

Specifically, cloud computing currently provides an interface to enable *on-demand self-service* and *rapid elasticity* [306]. The former means resource provisioning should be done on-demand and without human interaction with the service provider. The latter means the computation capability can be elastically provisioned and released. For cloud users, the computation resource available appears to be unlimited.

In the case of edge and fog computing, however, neither on-demand self-service nor rapid elasticity is available yet. Existing efforts [31, 60] have mainly considered the feasibility and benefit of edge computing compared to cloud computing. Therefore, these assume the edge server is a black box with unlimited computation resource, which is generally not the case in practice.

A further issue arises from the prospects of multiple control domains for edge and fog computing. While (public) cloud computing is mainly powered by a very small number of cloud service providers with extensive infrastructure deployment, edge and fog computing can be achieved with a range of infrastructure support, from a single cable modem to a city-scale micro data center [302]. There is concern that the latter could be expensive,

while the former may lack federation support. It is non-trivial to both find the nearest edge server and fully leverage the computation capacity across control domains.

Solution. Since edge and fog computing are considered essential for meeting increasingly interactive application demands, we explore how to automate basic management for the edge infrastructure. Specifically, we design an *edge management layer* to provide a unified topology management interface, such that even a single edge server can contribute resource by joining an existing edge network.

7.3 Challenges

In this section, we talk about the preliminary design to build the sharing economy model for the mobile edge, then we identify the design challenges and propose the corresponding solutions.

7.3.1 Preliminary design

The sharing economy model involves four parties: app vendors, app users, edge service providers and a Uber-like management system called Eber. For both app vendors and edge service providers, they are under different control domains.

Interactions between app users and app vendors. In order to attract users, app vendors are motivated to rent edge service in order to provide better user experience for their apps. Furthermore, depending on their business model, they may or may not include the edge service cost into the app subscription fees.

On the other hand, the app users are completely unaware of the existance of the edge service. However, when they use the apps, it will trigger the apps to send requests for edge service.

Interactions between edge service providers and Eber. Edge service providers, similar to cloud providers, maintain a cluster of edge servers which can provide computation and storage resource for the app vendors. The difference is, unlike cloud providers, they cannot directly communicate with app vendors; instead, they need to join a system similar to Uber or Airbnb, which can unify the edge server providers under various control domains.

In other words, any entity can provide the edge service by joining Eber.

For Eber, they manage these edge service providers and provide unified interfaces to app vendors. More specifically, they will accept offloading requests from apps, conduct the authentication check, schedule the requests to an edge server for computation, give the edge server credit and charge app vendors for renting the edge service as well as paying the edge service providers for providing the service.

Interactions between app vendors and Eber. For app vendors, they rent edge service and pay for it through Eber. However, a key difference is: for Uber scenario, passengers call vehicles through Uber for themselves; for the mobile edge computing senario, however, app vendors rent edge service not directly for themselves, but for their users. That means, it is the app user who initiates the edge service requests, not the app vendor.

7.3.2 Challenge 1: trust across different edge service providers

In the sharing economy model, edge servers are under different control domains, and they need to work together to provide service. That means they need to solve the trust issues across different domains, i.e., we need a privacy-preserving mechanism to protect the sensitive infrormation across edge servers under different control domains. Note that they don't trust Eber either, since they belong to different parties.

More specifically, the edge server providers want to protect the following information:

1. hardware specifications, including: hardware type (CPU, GPU or FPGA, etc.), edge server cluster setting (how many edge servers in the cluster, what is their network topology, what is the internal network setting, etc.), etc. Note that edge server providers may customize their hardware specs and have specific interconnect design so that they each can provide the fastest edge services to compete with each other, therefore it is natural for them to try to protect their hardware spec info;

2. workload information. Since edge servers can earn credits by processing workloads for the applications, the malicious nodes can infer edge service provider's revenue information if they know how much workload each edge server has processed. Note that if the edge server providers are individuals, they may not care about leaking their

revenue information. However, if a startup wants to provide the edge service, then the revenue information is confidential and must be protected;

7.3.3 Challenge 2: trust between edge servers and app vendors

App vendors need to offload part of their app code and data to edge servers, so it is natural for them to worry about curious edge service providers who try to probe their sensitive data, or malicious edge service providers who try to return them false results or even virus. On the other hand, for the edge service provider, they are worried about malicious app vendors who might upload malicious code to their edge servers to corrupt the machines or steal other app's data.

Therefore, the following challenges need to be addressed:

1. code isolation on edge servers. The edge service providers need to isolate app codes from different app vendors, and prevent app vendors from reading or writing other app's data or server OS's data.

2. encryption on user data. App vendors need to encrypt user data at runtime to prevent edge service providers from probing the data.

3. an auditing system which can verify the edge servers really execute app code and return the true result.

7.3.4 Challenge 3: clean interfaces for the app developers

Eber should be able to hide all the underlying complexities (multiple edge service providers under different control domains, different pricing models, QoS guarantees and service agreements, heterogeneous hardware, dynamic machine workload, etc.) and provide clean and easy-to-use interfaces to app vendors.

We think the following interfaces are needed:

1. an offloading interface. This interface should allow app vendors to offload part of their app code and app data to edge servers for processing. Unlike traditional offloading works which require the app vendors to specify the offloading destination, this interface

should be able to serve as an agent which can automatically do the matching between app vendors (codes) and edge servers.

2. a payment interface. This interface should collect workload info on each edge server (i.e., how long has each edge server work for app vendors) and generate bills for each app vendors. This is challenging because, the edge server providers may not want the Eber to know how long it has been working for the app vendors (because it involves the provider's revenue information), so we need to encrypt the edge server's workload information. In the meanwhile, edge server providers could cheat on this in order to get more credit, so a encryption protocol is needed to encrypt workload information and verify its correctness simultaneously.

7.3.5 Efficient resource provisioning mechanism

When apps initiate offloading requests to the edge servers, they need to provision certain amount of resource on the servers for remote processing. However, the resource provisioning is very challenging due to the following factors:

1. the app cannot know how much resource it needs for the offloading requests. It can always ask for a default amount of resource, e.g. ¡1core, 1GB memory¿, but this either leads to under-allocation which results in slow execution or even program failure (due to Out-Of-Memory errors), or results in over-allocation which wastes resource and money.

2. the edge servers could have resource fragments. We can treat the resource allocation problem as a bin-packing problem, which is NP-hard. Therefore, each edge servers will inevitably have resource fragments after resource allocation.

For the first issue, i.e., how to estimate app offloading requests' resource requirements, there are several approaches: collect app's history traces and analyze the "right" amount of resource allocation from the trace, or do complex code analysis to figure out the resource requirement of app's source code, or dynamically changes app's resource requirements based on its runtime behavior (this is suitable for long-running apps).

For the second issue, i.e., the resource fragment issue, existing works mainly focus on proposing greedy algorithm to minimize the resource fragments. However, since edge servers are under different control domains and have heterogeneous hardware settings (CPU, GPU, FPGA, etc.), the algorithm needs to be heterogeneous-aware and also needs to protect edge server's hardware setting info.

7.3.6 Service discovery

Service discovery can be divided two parts. One is how mobile devices discover available edge servers. The other is how edge servers that belong to different entities within the same area can discover each other. For the first problem, Universal Plug and Play (UPnP) technique mentioned in [31] provides a neat way to help mobile devices seamless discover nearby edge servers and get connected to them. The second one is the unique problem in our scenarios. We propose a neighbor discovery service to deal with this problem. Essentially, the neighbor discovery module constructs an overlay topology over the underlying IP network.

Although there is a large volume of work on self-organizing, scalable overlay network construction [307–309], we cannot incorporate an existing solution right away. This is because, in previous scenarios, each node maintains a list of neighbors they already knew due to the system setup. In contrast, an edge server in our scenario may have no way of knowing about other edge servers, if they are from different providers.

Therefore, bootstrapping the initial neighbor list is a unique challenge in our case. Once we address that, we can apply previous topology construction algorithms.

Bootstrapping. To bootstrap a standalone edge server, we propose a mobile topology dissemination technique. One key insight is to leverage the inherent mobility of IoT devices. Each IoT device stores the IP address of the last edge server it saw. When it connects to a new server via UPnP, it will inform the new server of last server IP address. This way, a standalone server can quickly get to know other edge servers in the system.

After that, the standalone server runs a peer sampling procedure [307] and, from the already known neighbor, receives a set of random neighbors in the network. This concludes the bootstrapping process.

The mobile topology dissemination technique incurs low overhead. Further, with high

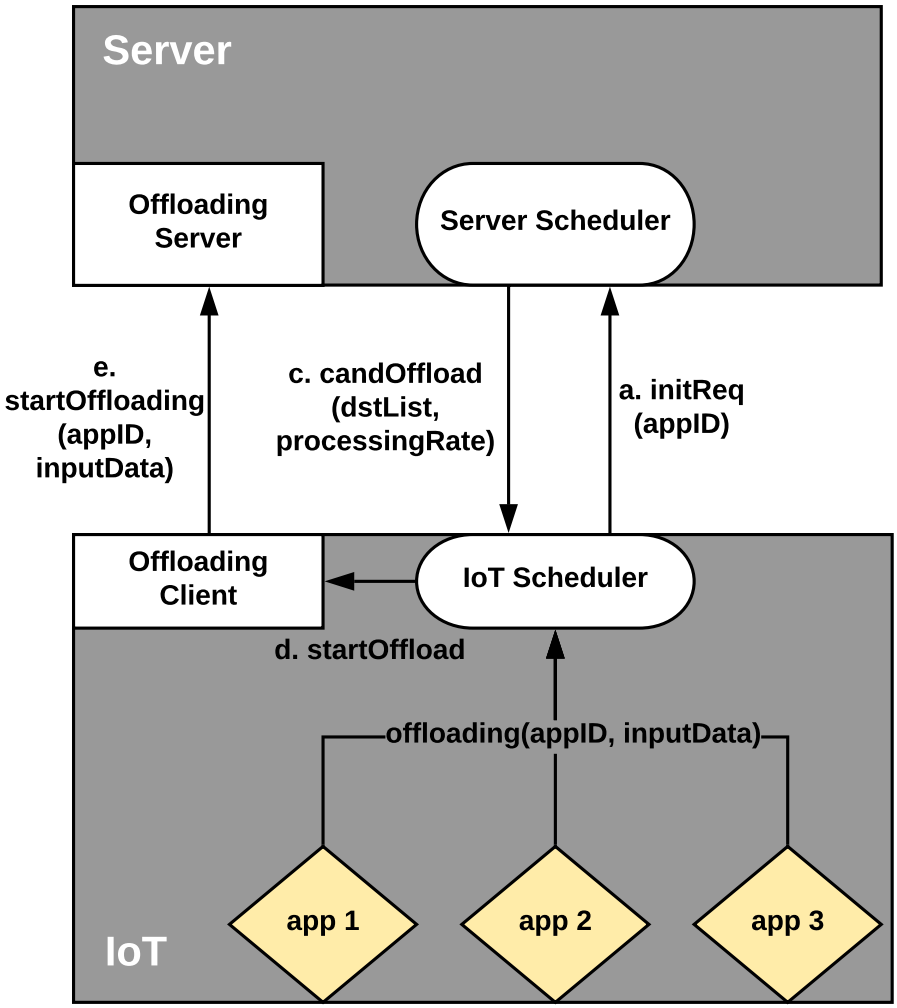


Figure 7.3: Scheduler Workflow.

probability, the last edge server seen by the IoT device is also near the current server, which heuristically makes the edge server tend to know the nearby server first.

Topology Construction. After bootstrapping, the problem becomes a typical overlay network construction problem. We run the T-Man [310] overlay network construction and maintenance protocol and use the ascending order of the RTT between different servers as the rank function in T-Man. The use of this gossip-based protocol results in fast convergence (logarithmically), high robustness in dynamic environments and local decision making.

7.3.7 Efficient scheduling

Figure 7.3 gives an overview of scheduling flow. We would like to help mobile devices to find out the optimized edge server in terms of end-to-end processing time.

Main components. Each IoT device runs two daemons, an IoT scheduler daemon and an offloading client daemon. The former sends scheduling requests to the server, while the latter sends the actual offloading job. They interact with their respective counterpart on the server, the server scheduler daemon and the offloading server daemon.

APIs. `offloading(appID, inputData)` will trigger the IoT scheduler to start a series of scheduling events. App developer needs to modify application's source code to directly call this API (only one line of code) to use our offloading service. `appID` is unique for each application.

There are also a few internal APIs between client-server daemon interaction:

`initReq(appID)` is used by the IoT scheduler daemon to connect to the server scheduler daemon, specifying the id of the application to be offloaded and asking for scheduling destination candidates.

`candOffload(dstList)` is used by the server scheduler daemon to return a list of scheduling destination candidates (decided by the scheduling logic) to the IoT scheduler.

`startOffloading(appID, inputData)` is used by the offloading client daemon to connect to the offloading server daemon on the final destination server, and send input data for remote processing.

Basic workflow. Each server runs a server-side scheduler daemon in a decentralized manner. The scheduler will record the processing rates for all offloading requests [1] executed on this server, and periodically share the rate information with its neighbors.

When an application wants to offload some computation, it will send an `offload` message to the IoT scheduler. The IoT scheduler will send an `initReq` message to the nearest server to ask for potential offloading destinations. The server scheduler tries to select the top two candidates with the highest processing rates for the app. If no servers processed this

1. The scheduler provides a default mechanism of estimating the application processing speed, i.e., the amount of input data being processed divided by the amount of time used. However, app developers can also provide customized estimation algorithms, e.g., taking skewness and contention into consideration.

application before (i.e., they do not have the processing rate information for the application), the server scheduler will select itself. The server scheduler sends a `candOffload` message back to the IoT scheduler with a list of potential destinations and their estimated processing rates.

From the received candidate list, the IoT scheduler estimates the total processing time on them as

$$T_{total} = \frac{S}{NetworkThroughput} + \frac{S}{R}$$

, where S is the size of the total input data for the session, and R is the estimated processing rate on the candidate server. Finally, the IoT scheduler chooses one with the least estimated total processing time, and notifies the offloading client daemon of the final destination.

The offloading client then sends an `startOffloading` message to the offloading server at the chosen destination, which will launch the corresponding Docker image based on the application id and process the input data. All offloading requests within the same session will always be sent to this same destination server.

7.4 Summary

In this chapter, we study the requirements for providing edge service, and propose a sharing economy model to organize the heterogeneous and decentralized edge servers. We further study the challenges for implementing such sharing economy model and talk about possible solutions.

Chapter 8

Conclusion and Future Directions

8.1 Conclusion

This thesis proposes a novel framework to simplify application development and deployment over a continuum of edge to cloud. We first highlight the overarching challenges of the application development process at the edge and propose a generic framework to address them, which consists of a set of new system abstractions. Specifically, *Crystal* masks hardware heterogeneity with abstract resource types through containerization and abstracts away the application processing pipelines into generic flow graphs. Further, it supports a notion of degradable computing for application scenarios at the edge that are involved with multi-modal sensory input (Chapter 4). *Video-zilla* is a generic data management service between video query systems and a video store to organize video data at the edge. We propose a video data unit abstraction based on the distance between objects in the video, quantifying the semantic similarity among video data (Chapter 5). Last, considering concurrent application execution, *LinkShare* supports multi-application offloading with device-centric control using a system-level scheduler service that wraps over the operating system scheduler (Chapter 6).

These three pieces of work sample the design space and provide key connections between different dimensions of design considerations, corresponding to the generic application development, data abstraction and resource management abstraction respectively. Altogether, we build a novel framework to enable easy application development, scalable data processing

and performance-aware resource management.

While building these systems, we gained a deeper understanding of applications at the edge and created an initial platform for easy application development and flexible system extension. Looking forward, we believe our framework can be further extended to use as a platform to explore the forthcoming application trends at the edge, for instance, asynchronous multi-stream data ingestion and data management for continuous deep learning.

8.2 Future directions

Designing system abstractions to achieve scalable application development at the edge is promising beyond this thesis. Edge computing is evolving fast. The edge infrastructure, edge workload, and data generated at the edge will become increasingly heterogeneous in the foreseeable future. Building future-proof system architectures is a must. Specifically, the following are several promising directions to explore.

Data management for continuous deep learning. In addition to the inference-based (e.g., face recognition, intrusion detection and speech recognition) workloads at the edge, there is a trend of continuous online learning (e.g. Federated learning [311]), where deep learning model training is distributed over the edge devices instead of in a centralized cloud. Much more data are needed for model training than the inference workload. Further, as model training relies more on the unseen data than seen data, a novel data abstraction and a data management system built on top of that are in need.

Asynchronous multi-stream data ingestion. By nature, data is generated at the edge asynchronously. For instance, a video camera on a mobile phone captures 30 video frames per second. At the same time, the microphone on the same mobile phone captures an audio signal with a 44.1 KHz sampling rate. For applications that process more than one input data stream, there is a lack of system-level support to synchronize these asynchronous data. Thus, current solutions are all application-specific. The right abstraction to enable asynchronous multi-stream data ingestion should not only cover existing data types but also be extendable to allow for emerging data types.

Bibliography

[1] Cellular iot and lpwa connectivity market tracker 2010-25. https://iot-analytics.com/product/cellular-iot-lpwa-connectivity-market-tracker-2020-25-update-q4-20/.

[2] C. Ericson. *Real-time collision detection*. Crc Press, 2004.

[3] J. Barthélemy, N. Verstaevel, H. Forehead, and P. Perez. Edge-computing video analytics for real-time traffic monitoring in a smart city. *Sensors*, 19(9):2048, 2019.

[4] S. Tanwar, P. Patel, K. Patel, S. Tyagi, N. Kumar, and M. S. Obaidat. An advanced internet of thing based security alert system for smart home. In *2017 international conference on computer, information and telecommunication systems (CITS)*, pages 25–29. IEEE, 2017.

[5] T. A. Khoa, L. M. B. Nhu, H. H. Son, N. M. Trong, C. H. Phuc, N. T. H. Phuong, N. Van Dung, N. H. Nam, D. S. T. Chau, and D. N. M. Duc. Designing efficient smart home management with iot smart lighting: a case study. *Wireless Communications and Mobile Computing*, 2020, 2020.

[6] Q. V. Vo, G. Lee, and D. Choi. Fall detection based on movement and smart phone technology. In *2012 IEEE RIVF International Conference on Computing & Communication Technologies, Research, Innovation, and Vision for the Future*, pages 1–4. IEEE, 2012.

[7] D. C. Yacchirema, D. Sarabia-Jácome, C. E. Palau, and M. Esteve. A smart system for sleep monitoring by integrating iot with big data analytics. *IEEE Access*, 6:35988–36001, 2018.

[8] R. C. Parpala and R. Iacob. Application of iot concept on predictive maintenance of industrial equipment. In *MATEC Web of Conferences*, volume 121, page 02008. EDP Sciences, 2017.

[9] Z. Shahbazi and Y.-C. Byun. Integration of blockchain, iot and machine learning for multistage quality control and enhancing security in smart manufacturing. *Sensors*, 21(4):1467, 2021.

[10] M. Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017.

[11] M. Satyanarayanan. How we created edge computing. *Nature Electronics*, 2(1):42–42, 2019.

[12] 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*, pages 49–62. ACM, 2010.

[13] 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*, pages 301–314. ACM, 2011.

[14] M. S. Gordon, D. A. Jamshidi, S. A. Mahlke, Z. M. Mao, and X. Chen. Comet: Code offload by migrating execution transparently. In *OSDI*, volume 12, pages 93–106, 2012.

[15] 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 *Infocom, 2012 Proceedings IEEE*, pages 945–953. IEEE, 2012.

[16] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the*

9th international conference on Mobile systems, applications, and services, pages 43–56. ACM, 2011.

[17] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.

[18] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[19] Raspberry pi os. `https://www.raspberrypi.com/software/`.

[20] Risc os. `https://www.riscosopen.org/content/`.

[21] Ubuntu core. `https://ubuntu.com/core`.

[22] Tensorflow 1.13. `https://github.com/tensorflow/tensorflow/releases/tag/v1.13.2`.

[23] Mxnet. `https://mxnet.apache.org/versions/1.8.0/`.

[24] Pytorch. `https://pytorch.org/`.

[25] R. Mitchell and R. Chen. A survey of intrusion detection in wireless network applications. *Computer Communications*, 42:1–23, 2014.

[26] 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*, pages 155–168. ACM, 2015.

[27] B. Hu and W. Hu. Linkshare: device-centric control for concurrent and continuous mobile-cloud interactions. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 15–29, 2019.

[28] White paper: Industrial transformation through the internet of things. `https://d1.awsstatic.com/product-marketing/iot-sitewise/IIoT-IDC-Whitepaper-US-Final.pdf`.

[29] Big data's role in the future of artificial intelligence across key verticals. https://go.abiresearch.com/lp-artificial-intelligence-meets-business-intelligence.

[30] Google tango project. https://get.google.com/tango/.

[31] 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*, pages 68–81. ACM, 2014.

[32] 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. 2017.

[33] IHS. Top video surveillance-trends-2018. https://cdn.ihs.com/www/pdf/Top-Video-Surveillance-Trends-2018.pdf.

[34] One surveillance camera for every 11 people in britain, says cctv survey. https://www.telegraph.co.uk/technology/10172298/One-surveillance-camera-for-every-11-people-in-Britain-says-CCTV-survey.html.

[35] C. Shan, F. Porikli, T. Xiang, and S. Gong. *Video analytics for business intelligence*. Springer, 2012.

[36] T. Zhang, A. Chowdhery, P. 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*, pages 426–438, 2015.

[37] S. P. Gulve, S. A. Khoje, and P. Pardeshi. Implementation of iot-based smart video surveillance system. In *Computational intelligence in data mining*, pages 771–780. Springer, 2017.

[38] A. C. Cob-Parro, C. Losada-Gutiérrez, M. Marrón-Romera, A. Gardel-Vicente, and I. Bravo-Muñoz. Smart video surveillance system based on edge computing. *Sensors*, 21(9):2958, 2021.

[39] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha. Real-time video analytics: The killer app for edge computing. *computer*, 50(10):58–67, 2017.

[40] T. Sikandar, K. H. Ghazali, and M. F. Rabbi. Atm crime detection using image processing integrated video surveillance: a systematic review. *Multimedia Systems*, 25(3):229–251, 2019.

[41] R. Sanders. Pursuing vision zero in seattle–results of a systemic safety analysis. 2016.

[42] B. A. Alpatov, P. V. Babayan, and M. D. Ershov. Vehicle detection and counting system for real-time traffic surveillance. In *2018 7th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–4. IEEE, 2018.

[43] H. Seibel, S. Goldenstein, and A. Rocha. Eyes on the target: Super-resolution and license-plate recognition in low-quality surveillance videos. *IEEE access*, 5:20020–20035, 2017.

[44] P. Tumas, A. Nowosielski, and A. Serackis. Pedestrian detection in severe weather conditions. *IEEE Access*, 8:62775–62784, 2020.

[45] E. S. Lohan, M. Koivisto, O. Galinina, S. Andreev, A. Tolli, G. Destino, M. Costa, K. Leppanen, Y. Koucheryavy, and M. Valkama. Benefits of positioning-aided communication technology in high-frequency industrial iot. *IEEE Communications Magazine*, 56(12):142–148, 2018.

[46] F. Foukalas and A. Tziouvaras. Edge ai for industrial iot applications. *IEEE Industrial Electronics Magazine*, 2021.

[47] Ibm business operations: Transform manufacturing operations for growth and subtainability. `https://www.ibm.com/business-operations/manufacturing`.

[48] Siemens industrial edge. https://siemens.mindsphere.io/en/industrial-iot/industrial-edge.

[49] P. Zheng, H. Wang, Z. Sang, R. Y. Zhong, Y. Liu, C. Liu, K. Mubarok, S. Yu, and X. Xu. Smart manufacturing systems for industry 4.0: Conceptual framework, scenarios, and future perspectives. *Frontiers of Mechanical Engineering*, 13(2):137–150, 2018.

[50] A. A. Süzen. A risk-assessment of cyber attacks and defense strategies in industry 4.0 ecosystem. *International Journal of Computer Network & Information Security*, 12(1), 2020.

[51] D. Z. Fawwaz and S.-H. Chung. Real-time and robust hydraulic system fault detection via edge computing. *Applied Sciences*, 10(17):5933, 2020.

[52] D. A. Rossit, F. Tohmé, and M. Frutos. An industry 4.0 approach to assembly line resequencing. *The International Journal of Advanced Manufacturing Technology*, 105(9):3619–3630, 2019.

[53] B. Asdecker and V. Felch. Development of an industry 4.0 maturity model for the delivery process in supply chains. *Journal of Modelling in Management*, 2018.

[54] G. Veiga, P. Malaca, and R. Cancela. Interactive industrial robot programming for the ceramic industry. *International Journal of Advanced Robotic Systems*, 10(10):354, 2013.

[55] Google lens. https://lens.google/.

[56] T. Hossain, M. S. Islam, M. A. R. Ahad, and S. Inoue. Human activity recognition using earable device. In *Adjunct Proceedings of the 2019 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2019 ACM International Symposium on Wearable Computers*, pages 81–84, 2019.

[57] R. R. Choudhury. Earable computing: A new area to think about. In *Proceedings of the 22nd International Workshop on Mobile Computing Systems and Applications*, pages 147–153, 2021.

[58] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal communications*, 8(4):10–17, 2001.

[59] R. K. Balan and J. Flinn. Cyber foraging: Fifteen years later. *IEEE Pervasive Computing*, 16(3):24–30, 2017.

[60] 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, 2009.

[61] Y. C. Hu, M. Patel, D. Sabella, N. Sprecher, and V. Young. Mobile edge computing—a key technology towards 5g. *ETSI white paper*, 11(11):1–16, 2015.

[62] F. Bonomi. Cloud and fog computing: Trade-offs and applications. In *NY: International Symposium on Computer Architecture*, 2011.

[63] R. Balan, J. Flinn, M. Satyanarayanan, S. Sinnamohideen, and H.-I. Yang. The case for cyber foraging. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 87–92. ACM, 2002.

[64] J. Flinn, D. Narayanan, and M. Satyanarayanan. Self-tuned remote execution for pervasive computing. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*, pages 61–66. IEEE, 2001.

[65] W. Zhu, C.-L. Wang, and F. C. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 381–388. IEEE, 2002.

[66] G. Lee, H. Park, S. Heo, K.-A. Chang, H. Lee, and H. Kim. Architecture-aware automatic computation offload for native applications. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 521–532. ACM, 2015.

[67] Z. Li, C. Wang, and R. Xu. Computation offloading to save energy on handheld devices: a partition scheme. In *Proceedings of the 2001 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 238–246. ACM, 2001.

[68] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 347–356. ACM, 2010.

[69] H. Flores and S. Srirama. Mobile code offloading: should it be a local decision or global inference? In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 539–540. ACM, 2013.

[70] R. Kemp, N. Palmer, T. Kielmann, and H. Bal. Cuckoo: a computation offloading framework for smartphones. In *International Conference on Mobile Computing, Applications, and Services*, pages 59–79. Springer, 2010.

[71] M. S. Gordon, D. K. Hong, P. M. Chen, J. Flinn, S. Mahlke, and Z. M. Mao. Accelerating mobile applications through flip-flop replication. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 137–150. ACM, 2015.

[72] 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*, pages 123–136. ACM, 2016.

[73] F. A. Salaht, F. Desprez, and A. Lebre. An overview of service placement problem in fog and edge computing. *ACM Computing Surveys (CSUR)*, 53(3):1–35, 2020.

[74] R. Mahmud, R. Kotagiri, and R. Buyya. Fog computing: A taxonomy, survey and future directions. In *Internet of everything*, pages 103–130. Springer, 2018.

[75] C.-C. Hung, G. Ananthanarayanan, P. Bodik, L. Golubchik, M. Yu, P. Bahl, and M. Philipose. Videoedge: Processing camera streams using hierarchical clusters. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 115–131. IEEE, 2018.

[76] M. Taneja and A. Davy. Resource aware placement of iot application modules in fog-cloud computing paradigm. In *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 1222–1228. IEEE, 2017.

[77] Y. Xia, X. Etchevers, L. Letondeur, T. Coupaye, and F. Desprez. Combining hardware nodes and software components ordering-based heuristics for optimizing the placement of distributed iot applications in the fog. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 751–760, 2018.

[78] P. Wang, S. Liu, F. Ye, and X. Chen. A fog-based architecture and programming model for iot applications in the smart grid. *arXiv preprint arXiv:1804.01239*, 2018.

[79] O. Skarlat, M. Nardelli, S. Schulte, and S. Dustdar. Towards qos-aware fog service placement. In *2017 IEEE 1st international conference on Fog and Edge Computing (ICFEC)*, pages 89–96. IEEE, 2017.

[80] C. Perera, Y. Qin, J. C. Estrella, S. Reiff-Marganiec, and A. V. Vasilakos. Fog computing for sustainable smart cities: A survey. *ACM Computing Surveys (CSUR)*, 50(3):1–43, 2017.

[81] J. Santos, T. Wauters, B. Volckaert, and F. De Turck. Resource provisioning for iot application services in smart cities. In *2017 13th International Conference on Network and Service Management (CNSM)*, pages 1–9. IEEE, 2017.

[82] S. Sarkar, S. Chatterjee, and S. Misra. Assessment of the suitability of fog computing in the context of internet of things. *IEEE Transactions on Cloud Computing*, 6(1):46–59, 2015.

[83] S. Sarkar and S. Misra. Theoretical modelling of fog computing: a green computing paradigm to support iot applications. *Iet Networks*, 5(2):23–29, 2016.

[84] H. R. Arkian, A. Diyanat, and A. Pourkhalili. Mist: Fog-based data analytics scheme with cost-efficient resource provisioning for iot crowdsensing applications. *Journal of Network and Computer Applications*, 82:152–165, 2017.

[85] H.-J. Hong, P.-H. Tsai, A.-C. Cheng, M. Y. S. Uddin, N. Venkatasubramanian, and C.-H. Hsu. Supporting internet-of-things analytics in a fog computing platform. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 138–145. IEEE, 2017.

[86] I. Alam, K. Sharif, F. Li, Z. Latif, M. M. Karim, S. Biswas, B. Nour, and Y. Wang. A survey of network virtualization techniques for internet of things using sdn and nfv. *ACM Computing Surveys (CSUR)*, 53(2):1–40, 2020.

[87] R. Morabito. Virtualization on internet of things edge devices with container technologies: A performance evaluation. *IEEE Access*, 5:8835–8850, 2017.

[88] R. Morabito et al. Lightweight virtualization in edge computing for internet of things. 2019.

[89] R. P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, 1974.

[90] P. Sharma, L. Chaufournier, P. Shenoy, and Y. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, pages 1–13, 2016.

[91] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News*, 41(1):461–472, 2013.

[92] R. Morabito, V. Cozzolino, A. Y. Ding, N. Beijar, and J. Ott. Consolidate iot edge computing with lightweight virtualization. *Ieee network*, 32(1):102–111, 2018.

[93] I. Stojmenovic. Fog computing: A cloud to the ground support for smart things and machine-to-machine networks. In *2014 Australasian telecommunication networks and applications conference (ATNAC)*, pages 117–122. IEEE, 2014.

[94] B. Amento, B. Balasubramanian, R. J. Hall, K. Joshi, G. Jung, and K. H. Purdy. Focusstack: Orchestrating edge clouds using location-based focus of attention. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 179–191. IEEE, 2016.

[95] H.-J. Hong, J.-C. Chuang, and C.-H. Hsu. Animation rendering on multimedia fog computing platforms. In *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 336–343. IEEE, 2016.

[96] A. Gosain, M. Berman, M. Brinn, T. Mitchell, C. Li, Y. Wang, H. Jin, J. Hua, and H. Zhang. Enabling campus edge computing using geni racks and mobile resources. In *2016 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 41–50. IEEE, 2016.

[97] A. C. Baktir, A. Ozgovde, and C. Ersoy. How can edge computing benefit from software-defined networking: A survey, use cases, and future directions. *IEEE Communications Surveys & Tutorials*, 19(4):2359–2391, 2017.

[98] G. Orsini, D. Bade, and W. Lamersdorf. Cloudaware: A context-adaptive middleware for mobile edge and cloud computing applications. In *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS* W)*, pages 216–221. IEEE, 2016.

[99] J. Rodrigues, E. R. Marques, L. M. Lopes, and F. Silva. Towards a middleware for mobile edge-cloud applications. In *Proceedings of the 2nd workshop on middleware for Edge Clouds & Cloudlets*, pages 1–6, 2017.

[100] A. Carrega, M. Repetto, P. Gouvas, and A. Zafeiropoulos. A middleware for mobile edge computing. *IEEE Cloud Computing*, 4(4):26–37, 2017.

[101] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, et al. Lessons learned from the chameleon testbed. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 219–233, 2020.

[102] K. Keahey, J. Anderson, M. Sherman, Z. Zhen, M. Powers, I. Brunkan, and A. Cooper. Chameleon@ edge community workshop report. 2021.

[103] E. Piri, P. Ruuska, T. Kanstren, J. Mäkelä, J. Korva, A. Hekkala, A. Pouttu, O. Liinamaa, M. Latva-Aho, K. Vierimaa, et al. 5gtn: A test network for 5g application development and testing. In *2016 European Conference on Networks and Communications (EuCNC)*, pages 313–318. IEEE, 2016.

[104] A. Gosain. Platforms for advanced wireless research: Helping define a new edge computing paradigm. In *Proceedings of the 2018 on Technologies for the Wireless Edge Workshop*, pages 33–33, 2018.

[105] D. Raychaudhuri, I. Seskar, G. Zussman, T. Korakis, D. Kilper, T. Chen, J. Kolodziejski, M. Sherman, Z. Kostic, X. Gu, et al. Challenge: Cosmos: A city-scale programmable testbed for experimentation with advanced wireless. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–13, 2020.

[106] J. Breen, A. Buffmire, J. Duerig, K. Dutt, E. Eide, A. Ghosh, M. Hibler, D. Johnson, S. K. Kasera, E. Lewis, et al. Powder: Platform for open wireless data-driven experimental research. *Computer Networks*, 197:108281, 2021.

[107] A. Panicker, O. Ozdemir, M. L. Sichitiu, I. Guvenc, R. Dutta, V. Marojevic, and B. Floyd. Aerpaw emulation overview and preliminary performance evaluation. *Computer Networks*, 194:108083, 2021.

[108] H. Zhang, Y. Guan, A. Kamal, D. Qiao, M. Zheng, A. Arora, O. Boyraz, B. Cox, T. Daniels, M. Darr, et al. Ara: A wireless living lab vision for smart and connected rural communities. In *Proceedings of the 15th ACM Workshop on Wireless Network Testbeds, Experimental evaluation & CHaracterization*, pages 9–16, 2022.

[109] Connected cars use case for mobile edge computing. `https://www.nokia.com/networks/portfolio/edge-cloud/`.

[110] Y. Xiao, Y. Jia, C. Liu, X. Cheng, J. Yu, and W. Lv. Edge computing security: State of the art and challenges. *Proceedings of the IEEE*, 107(8):1608–1631, 2019.

[111] J. Zhang, B. Chen, Y. Zhao, X. Cheng, and F. Hu. Data security and privacy-preserving in edge computing paradigm: Survey and open issues. *IEEE access*, 6:18209–18237, 2018.

[112] D. Balfanz, D. K. Smetters, P. Stewart, and H. C. Wong. Talking to strangers: Authentication in ad-hoc wireless networks. In *NDSS*. Citeseer, 2002.

[113] S. Bouzefrane, A. F. B. Mostefa, F. Houacine, and H. Cagnon. Cloudlets authentication in nfc-based mobile computing. In *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*, pages 267–272. IEEE, 2014.

[114] I. Stojmenovic, S. Wen, X. Huang, and H. Luan. An overview of fog computing and its security issues. *Concurrency and Computation: Practice and Experience*, 28(10):2991–3005, 2016.

[115] K. Yang and X. Jia. Data storage auditing service in cloud computing: challenges, methods and opportunities. *World Wide Web*, 15(4):409–428, 2012.

[116] C. Wang, Q. Wang, K. Ren, and W. Lou. Privacy-preserving public auditing for data storage security in cloud computing. In *2010 proceedings ieee infocom*, pages 1–9. Ieee, 2010.

[117] Q. Wang, C. Wang, K. Ren, W. Lou, and J. Li. Enabling public auditability and data dynamics for storage security in cloud computing. *IEEE transactions on parallel and distributed systems*, 22(5):847–859, 2010.

[118] K. Yang and X. Jia. An efficient and secure dynamic auditing protocol for data storage in cloud computing. *IEEE transactions on parallel and distributed systems*, 24(9):1717–1726, 2012.

[119] C. Lin, Z. Shen, Q. Chen, and F. T. Sheldon. A data integrity verification scheme in mobile cloud computing. *Journal of Network and Computer Applications*, 77:146–151, 2017.

[120] R. Gennaro, C. Gentry, and B. Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Annual Cryptology Conference*, pages 465–482. Springer, 2010.

[121] C. Wang, N. Cao, K. Ren, and W. Lou. Enabling secure and efficient ranked keyword search over outsourced cloud data. *IEEE Transactions on parallel and distributed systems*, 23(8):1467–1479, 2011.

[122] A.-A. Ivan and Y. Dodis. Proxy cryptography revisited. In *NDSS*. Citeseer, 2003.

[123] B. R. Waters, D. Balfanz, G. Durfee, and D. K. Smetters. Building an encrypted and searchable audit log. In *NDSS*, volume 4, pages 5–6, 2004.

[124] Common objects in context dataset.

[125] Security cameras - security, chicago transit authority. `https://www.transitchicago.com/security/cameras/`.

[126] J utah. `https://www.youtube.com/channel/UCBcVQr-07MH-p9e2kRTdB3A/`.

[127] blessedhomosapiens. American roads. `https://www.youtube.com/user/blessedhomosapiens/videos`.

[128] V. Railfan. Kansas city east. `https://www.youtube.com/watch?v=WUBQJosNm8o`.

[129] A. Tech. Werkhaven urk. `https://www.youtube.com/watch?v=IebzjqLQoYg`.

[130] K. A. Joshi and D. G. Thakore. A survey on moving object detection and tracking in video surveillance system. *International Journal of Soft Computing and Engineering*, 2(3):44–48, 2012.

[131] S. He, H. Luo, P. Wang, F. Wang, H. Li, and W. Jiang. Transreid: Transformer-based object re-identification. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 15013–15022, 2021.

[132] M. De-la Torre, E. Granger, P. V. Radtke, R. Sabourin, and D. O. Gorodnichy. Partially-supervised learning from facial trajectories for face recognition in video surveillance. *Information fusion*, 24:31–53, 2015.

[133] J. Cheng, W. Chen, F. Tao, and C.-L. Lin. Industrial iot in 5g environment towards smart manufacturing. *Journal of Industrial Information Integration*, 10:10–19, 2018.

[134] P. Guo, B. Hu, and W. Hu. Mistify: Automating dnn model porting for on-device inference at the edge. In *NSDI*, pages 705–719, 2021.

[135] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[136] J. Ngiam, A. Khosla, M. Kim, J. Nam, H. Lee, and A. Y. Ng. Multimodal deep learning. In *ICML*, 2011.

[137] J. Gao, P. Li, Z. Chen, and J. Zhang. A survey on deep learning for multimodal data fusion. *Neural Computation*, 32(5):829–864, 2020.

[138] Y. Mroueh, E. Marcheret, and V. Goel. Deep multimodal learning for audio-visual speech recognition. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2130–2134. IEEE, 2015.

[139] Covarep: A cooperative voice analysis repository for speech technologies. `https://github.com/covarep/covarep`.

[140] Openface 2.2.0: a facial behavior analysis toolkit. `https://github.com/TadasBaltrusaitis/OpenFace`.

[141] C. Dixon, R. Mahajan, S. Agarwal, A. Brush, B. Lee, S. Saroiu, and V. Bahl. The home needs an operating system (and an app store). In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, pages 1–6, 2010.

[142] V. Saligrama and Z. Chen. Video anomaly detection based on local statistical aggregates. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2112–2119. IEEE, 2012.

[143] N. Almeida, S. Silva, A. Teixeira, M. Ketsmur, D. Guimarães, and E. Fonseca. Multimodal interaction for accessible smart homes. In *Proceedings of the 8th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion*, pages 63–70, 2018.

[144] H. Boyes, B. Hallaq, J. Cunningham, and T. Watson. The industrial internet of things (iiot): An analysis framework. *Computers in industry*, 101:1–12, 2018.

[145] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[146] K. S. Oie, T. Kiemel, and J. J. Jeka. Multisensory fusion: simultaneous re-weighting of vision and touch for the control of human posture. *Cognitive Brain Research*, 14(1):164–176, 2002.

[147] T. Baltrušaitis, C. Ahuja, and L.-P. Morency. Multimodal machine learning: A survey and taxonomy. *IEEE transactions on pattern analysis and machine intelligence*, 41(2):423–443, 2018.

[148] 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*, pages 613–627, 2017.

[149] N. Ruiz, R. Taib, and F. Chen. Examining the redundancy of multimodal input. In *Proceedings of the 18th Australia conference on Computer-Human Interaction: Design: Activities, Artefacts and Environments*, pages 389–392, 2006.

[150] A. Jaimes and N. Sebe. Multimodal human–computer interaction: A survey. *Computer vision and image understanding*, 108(1-2):116–134, 2007.

[151] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han. Once-for-all: Train one network and specialize it for efficient deployment. *arXiv preprint arXiv:1908.09791*, 2019.

[152] Y. Lee, A. Scolari, B.-G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi. {PRETZEL}: Opening the black box of machine learning prediction serving systems. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 611–626, 2018.

[153] 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 *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 97–112, 2014.

[154] Kubernetes. https://kubernetes.io/.

[155] Docker. https://www.docker.com/.

[156] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News*, 45(1):615–629, 2017.

[157] S. Teerapittayanon, B. McDanel, and H.-T. Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2464–2469. IEEE, 2016.

[158] S. Laskaridis, S. I. Venieris, M. Almeida, I. Leontiadis, and N. D. Lane. Spinn: synergistic progressive inference of neural networks over device and cloud. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, pages 1–15, 2020.

[159] Protocol buffers — google developers. https://developers.google.com/protocol-buffers.

[160] Introducing json. https://www.json.org/json-en.html.

[161] The annotated xml specification. https://www.xml.com/axml/axml.html.

[162] M. Zhu and S. Gupta. To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*, 2017.

[163] Boost c++ libraries. https://www.boost.org/.

[164] Java native interface. https://docs.oracle.com/javase/8/docs/technotes/guides/jni/.

[165] Pybind11 documentation. https://pybind11.readthedocs.io/en/stable/.

[166] C++ mex applications. https://www.mathworks.com/help/matlab/cpp-mex-file-applications.html.

[167] Microsoft neural network intelligence toolkit. https://github.com/microsoft/nni.

[168] librosa: a python package for music and audio analysis. https://github.com/covarep/covarep.

[169] Essentia: an open-source c++ library for audio analysis and audio-based music information retrieval. https://github.com/MTG/essentia.

[170] Word2vec. https://code.google.com/archive/p/word2vec/.

[171] J. Pennington, R. Socher, and C. D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[172] Huggingface: The ai community building the future. https://huggingface.co/.

[173] Docker swarm. https://docs.docker.com/engine/swarm/.

[174] A. Zadeh, P. P. Liang, N. Mazumder, S. Poria, E. Cambria, and L.-P. Morency. Memory fusion network for multi-view sequential learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[175] T. Kucherenko, P. Jonell, S. van Waveren, G. E. Henter, S. Alexandersson, I. Leite, and H. Kjellström. Gesticulator: A framework for semantically-aware speech-driven gesture generation. In *Proceedings of the 2020 International Conference on Multimodal Interaction*, pages 242–250, 2020.

[176] I. Calixto, Q. Liu, and N. Campbell. Incorporating global visual features into attention-based neural machine translation. *arXiv preprint arXiv:1701.06521*, 2017.

[177] A. Zadeh, P. P. Liang, S. Poria, P. Vij, E. Cambria, and L.-P. Morency. Multi-attention recurrent network for human communication comprehension. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[178] D. Elliott, S. Frank, K. Sima'an, and L. Specia. Multi30k: Multilingual english-german image descriptions. *arXiv preprint arXiv:1605.00459*, 2016.

[179] T. Gupta, R. P. Singh, A. Phanishayee, J. Jung, and R. Mahajan. Bolt: Data management for connected homes. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 243–256, 2014.

[180] S. Kumar, Y. Hu, M. P. Andersen, R. A. Popa, and D. E. Culler. {JEDI}: Many-to-many end-to-end encryption and key delegation for iot. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1519–1536, 2019.

[181] Apple homekit. `https://developer.apple.com/homekit/`.

[182] Home assistant: Open source home automation that puts local control and privacy first. `https://www.home-assistant.io/`.

[183] Aws iot things graph. `https://aws.amazon.com/iot-thingsgraph/`.

[184] Edgex foundry. `https://www.edgexfoundry.org/`.

[185] Samsung smartthings. `https://smartthings.developer.samsung.com/`.

[186] S. Poria, E. Cambria, R. Bajpai, and A. Hussain. A review of affective computing: From unimodal analysis to multimodal fusion. *Information Fusion*, 37:98–125, 2017.

[187] P. Ledin and D. Machin. *Introduction to multimodal analysis*. Bloomsbury Publishing, 2020.

[188] S. Mai, H. Hu, and S. Xing. Divide, conquer and combine: Hierarchical feature fusion network with local and global perspectives for multimodal affective computing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 481–492, 2019.

[189] S. Palaskar, R. Sanabria, and F. Metze. End-to-end multimodal speech recognition. In *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5774–5778. IEEE, 2018.

[190] Affective computing market. `https://www.marketsandmarkets.com/Market-Reports/affective-computing-market-130730395.html`.

[191] T. Li, J. Huang, E. Risinger, and D. Ganesan. Low-latency speculative inference on distributed multi-modal data streams. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 67–80, 2021.

[192] H. Röger and R. Mayer. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Computing Surveys (CSUR)*, 52(2):1–37, 2019.

[193] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 783–798, 2018.

[194] X. Fu, T. Ghaffar, J. C. Davis, and D. Lee. Edgewise: a better stream processing engine for the edge. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 929–946, 2019.

[195] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram. Nexus: a gpu cluster engine for accelerating dnn-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 322–337, 2019.

[196] D. Crankshaw, G.-E. Sela, X. Mo, C. Zumar, I. Stoica, J. Gonzalez, and A. Tumanov. Inferline: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 477–491, 2020.

[197] A. Boroumand, S. Ghose, B. Akin, R. Narayanaswami, G. F. Oliveira, X. Ma, E. Shiu, and O. Mutlu. Mitigating edge machine learning inference bottlenecks: An empirical study on accelerating google edge models. *arXiv preprint arXiv:2103.00768*, 2021.

[198] W. Zhang, Q. J. Wu, and H. bing Yin. Moving vehicles detection based on adaptive motion histogram. *Digital Signal Processing*, 20(3):793–805, 2010.

[199] R. T. Collins, A. J. Lipton, H. Fujiyoshi, and T. Kanade. Algorithms for cooperative multisensor surveillance. *Proceedings of the IEEE*, 89(10):1456–1477, 2001.

[200] Z. Zhou, X. Chen, Y.-C. Chung, Z. He, T. X. Han, and J. M. Keller. Activity analysis, summarization, and visualization for indoor human activity monitoring. *IEEE Transactions on Circuits and Systems for Video Technology*, 18(11):1489–1498, 2008.

[201] M. Xu, T. Xu, Y. Liu, X. Liu, G. Huang, and F. X. Lin. Supporting video queries on zero-streaming cameras. *arXiv preprint arXiv:1904.12342*, 2019.

[202] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. Noscope: optimizing neural network queries over video at scale. *Proceedings of the VLDB Endowment*, 10(11):1586–1597, 2017.

[203] S. Jain, G. Ananthanarayanan, J. Jiang, Y. Shu, and J. Gonzalez. Scaling video analytics systems to large camera deployments. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, pages 9–14, 2019.

[204] K. Hsieh, G. Ananthanarayanan, P. Bodik, S. Venkataraman, P. Bahl, M. Philipose, P. B. Gibbons, and O. Mutlu. Focus: Querying large video datasets with low latency and low cost. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 269–286, 2018.

[205] S. Jain, X. Zhang, Y. Zhou, G. Ananthanarayanan, J. Jiang, Y. Shu, P. Bahl, and J. Gonzalez. Spatula: Efficient cross-camera video analytics on large camera networks. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 110–124. IEEE, 2020.

[206] S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1579–1590, 2014.

[207] F. Abuzaid, P. Kraft, S. Suri, E. Gan, E. Xu, A. Shenoy, A. Ananthanarayan, J. Sheu, E. Meijer, X. Wu, et al. Diff: a relational interface for large-scale data explanation. *Proceedings of the VLDB Endowment*, 12(4):419–432, 2018.

[208] S. Roy, A. C. König, I. Dvorkin, and M. Kumar. Perfaugur: Robust diagnostics for performance anomalies in cloud services. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1167–1178. IEEE, 2015.

[209] D. Y. Yoon, N. Niu, and B. Mozafari. Dbsherlock: A performance diagnostic tool for transactional databases. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1599–1614, 2016.

[210] Jackson hole webcam. `https://www.seejh.com/webcams/jackson`.

[211] S. Oh, A. Hoogs, A. Perera, N. Cuntoor, C.-C. Chen, J. T. Lee, S. Mukherjee, J. Aggarwal, H. Lee, L. Davis, et al. A large-scale benchmark dataset for event recognition in surveillance video. In *CVPR 2011*, pages 3153–3160. IEEE, 2011.

[212] S. Chinoy. We built an 'unbelievable' (but legal) facial recognition machine, Apr 2019.

[213] A. Clapés, M. Reyes, and S. Escalera. Multi-modal user identification and object recognition surveillance system. *Pattern Recognition Letters*, 34(7):799–808, 2013.

[214] G. Diamantopoulos and M. Spann. Event detection for intelligent car park video surveillance. *Real-Time Imaging*, 11(3):233–243, 2005.

[215] National oceanic and atmospheric administration surveillance market report. `https://cdn.exacq.com/auto/casestudy/pdf/a31caaeb-9a6f-f464-4590-08d97fff9828.pdf?rand=0.11452011740766466`.

[216] J. Redmon and A. Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[217] L. Liu, H. Li, and M. Gruteser. Edge assisted real-time object detection for mobile augmented reality. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–16, 2019.

[218] V. Ruzicka and F. Franchetti. Fast and accurate object detection in high resolution 4k and 8k video using gpus. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[219] W. Zhang, Z. He, L. Liu, Z. Jia, Y. Liu, M. Gruteser, D. Raychaudhuri, and Y. Zhang. Elf: accelerate high-resolution mobile deep vision with content-aware parallel offloading. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, pages 201–214, 2021.

[220] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[221] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[222] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2117–2125, 2017.

[223] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[224] Y. Rubner, C. Tomasi, and L. J. Guibas. The earth mover's distance as a metric for image retrieval. *International journal of computer vision*, 40(2):99–121, 2000.

[225] H. Ling and K. Okada. An efficient earth mover's distance algorithm for robust histogram comparison. *IEEE transactions on pattern analysis and machine intelligence*, 29(5):840–853, 2007.

[226] O. Pele and M. Werman. Fast and robust earth mover's distances. In *2009 IEEE 12th International Conference on Computer Vision*, pages 460–467. IEEE, 2009.

[227] Y. Rubner, C. Tomasi, and L. J. Guibas. A metric for distributions with applications to image databases. In *Sixth International Conference on Computer Vision (IEEE Cat. No. 98CH36271)*, pages 59–66. IEEE, 1998.

[228] O. Pele and M. Werman. A linear time histogram metric for improved sift matching. In *European conference on computer vision*, pages 495–508. Springer, 2008.

[229] P. J. Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[230] A. Kobren, N. Monath, A. Krishnamurthy, and A. McCallum. A hierarchical algorithm for extreme clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 255–264, 2017.

[231] K. A. Heller and Z. Ghahramani. Bayesian hierarchical clustering. In *Proceedings of the 22nd international conference on Machine learning*, pages 297–304, 2005.

[232] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[233] M. Verleysen and D. François. The curse of dimensionality in data mining and time series prediction. In *International work-conference on artificial neural networks*, pages 758–770. Springer, 2005.

[234] E. Chávez and G. Navarro. Probabilistic proximity search: Fighting the curse of dimensionality in metric spaces. *Information Processing Letters*, 85(1):39–46, 2003.

[235] Microsoft rocket video analytics platform. `https://github.com/microsoft/Microsoft-Rocket-Video-Analytics-Platform`.

[236] Keras availabel models. `https://keras.io/applications/`.

[237] B. Rinner. Can we trust smart cameras? *Computer*, 52(5):67–70, 2019.

[238] Akka toolkit. `https://akka.io/`.

[239] Keras, 2.3.1. `https://github.com/keras-team/keras/releases/tag/2.3.1`.

[240] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer. Moa: Massive online analysis. *Journal of Machine Learning Research*, 11(May):1601–1604, 2010.

[241] Jfastemd. `https://github.com/telmomenezes/JFastEMD`.

[242] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter. Sprocket: A serverless video processing framework. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 263–274, 2018.

[243] Q. Huang, P. Ang, P. Knowles, T. Nykiel, I. Tverdokhlib, A. Yajurvedi, P. Dapolito IV, X. Yan, M. Bykov, C. Liang, et al. Sve: Distributed video processing at facebook scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 87–103, 2017.

[244] V. Railfan. San juan capistrano, california usa - virtual railfan live. `https://www.youtube.com/watch?v=zXqx56hlpd4`.

[245] WebCamNL. Webcam.nl — www.hollandmarinas.com — live full hd ptz camera. `https://www.youtube.com/watch?v=7g0lWkV61lY&feature=emb_logo`.

[246] D. Müllner. Modern hierarchical, agglomerative clustering algorithms. *arXiv preprint arXiv:1109.2378*, 2011.

[247] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Vldb*, volume 97, pages 426–435. Citeseer, 1997.

[248] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *Proceedings of the 20th international conference on World wide web*, pages 577–586, 2011.

[249] Benchmarking nearest neighbors. `https://github.com/erikbern/ann-benchmarks`.

[250] Pynndescent. `https://github.com/lmcinnes/pynndescent`.

[251] W. Hu, N. Xie, L. Li, X. Zeng, and S. Maybank. A survey on visual content-based video indexing and retrieval. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 41(6):797–819, 2011.

[252] M. S. Lew, N. Sebe, C. Djeraba, and R. Jain. Content-based multimedia information retrieval: State of the art and challenges. *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, 2(1):1–19, 2006.

[253] C. G. Snoek, M. Worring, et al. Concept-based video retrieval. *Foundations and Trends® in Information Retrieval*, 2(4):215–322, 2009.

[254] J. Yuan, H. Wang, L. Xiao, W. Zheng, J. Li, F. Lin, and B. Zhang. A formal study of shot boundary detection. *IEEE transactions on circuits and systems for video technology*, 17(2):168–186, 2007.

[255] X. Li, B. Zhao, and X. Lu. Key frame extraction in the summary space. *IEEE transactions on cybernetics*, 48(6):1923–1934, 2017.

[256] S.-F. Chang, W.-Y. Ma, and A. Smeulders. Recent advances and challenges of semantic image/video search. In *2007 IEEE International Conference on Acoustics, Speech and Signal Processing-ICASSP'07*, volume 4, pages IV–1205. IEEE, 2007.

[257] W. Ren, S. Singh, M. Singh, and Y. Zhu. State-of-the-art on spatio-temporal information-based video retrieval. *Pattern Recognition*, 42(2):267–282, 2009.

[258] X. Liu, P. Ghosh, O. Ulutan, B. Manjunath, K. Chan, and R. Govindan. Caesar: cross-camera complex activity recognition. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, pages 232–244, 2019.

[259] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. In *Software pioneers*, pages 245–262. Springer, 2002.

[260] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196. ACM, 2012.

[261] D. Yuan and P. Mitra. Lindex: a lattice-based index for graph databases. *The VLDB Journal—The International Journal on Very Large Data Bases*, 22(2):229–252, 2013.

[262] M. R. Anderson, M. Cafarella, G. Ros, and T. F. Wenisch. Physical representation-based predicate optimization for a visual analytics database. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1466–1477. IEEE, 2019.

[263] T. Xu, L. M. Botelho, and F. X. Lin. Vstore: A data store for analytics on large videos. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17, 2019.

[264] B. Haynes, M. Daum, D. He, A. Mazumdar, M. Balazinska, A. Cheung, and L. Ceze. Vss: A storage system for video analytics. In *Proceedings of the 2021 International Conference on Management of Data*, pages 685–696, 2021.

[265] D. Kang, P. Bailis, and M. Zaharia. Blazeit: optimizing declarative aggregation and limit queries for neural network-based video analytics. *arXiv preprint arXiv:1805.01046*, 2018.

[266] D. Kang, J. Guibas, P. Bailis, T. Hashimoto, and M. Zaharia. Task-agnostic indexes for deep learning-based queries over unstructured data. *arXiv preprint arXiv:2009.04540*, 2020.

[267] B. Haynes, A. Mazumdar, M. Balazinska, L. Ceze, and A. Cheung. Lightdb: A dbms for virtual reality video. *Proceedings of the VLDB Endowment*, 11(10), 2018.

[268] B. Haynes, M. Daum, A. Mazumdar, M. Balazinska, A. Cheung, and L. Ceze. Visualworlddb: A dbms for the visual world. In *CIDR*, 2020.

[269] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[270] 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*, page 15. ACM, 2017.

[271] 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*, pages 57–70, 2016.

[272] A. Poms, W. Crichton, P. Hanrahan, and K. Fatahalian. Scanner: Efficient video analysis at scale. *ACM Transactions on Graphics (TOG)*, 37(4):138, 2018.

[273] F. Bastani, S. He, A. Balasingam, K. Gopalakrishnan, M. Alizadeh, H. Balakrishnan, M. Cafarella, T. Kraska, and S. Madden. Miris: fast object track queries in video. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1907–1921, 2020.

[274] H. Vogt. Efficient object identification with passive rfid tags. In *International Conference on Pervasive Computing*, pages 98–113. Springer, 2002.

[275] M. Satyanarayanan. From the editor in chief: Augmenting cognition. *IEEE Pervasive Computing*, 3(2):4–5, 2004.

[276] A. Hampapur, L. Brown, J. Connell, A. Ekin, N. Haas, M. Lu, H. Merkl, and S. Pankanti. Smart video surveillance: exploring the concept of multiscale spatiotemporal tracking. *IEEE Signal Processing Magazine*, 22(2):38–51, 2005.

[277] Ip video surveillance and vsaas market by type (ip camera, monitors, storage, vms, video analytics, cloud storage by product software, cloud storage by deployment, vsaas, hosted vsaas, managed vsaas, hybrid vsaas, integration services) and application (banking & financial, retail, healthcare, government & higher, security, manufacturing & corporate, residential, entertainment & casino) - global opportunity analysis and industry forecast, 2015 - 2022, 2016.

[278] 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 *NSDI*, volume 9, page 1, 2017.

[279] T. J. Hacker, B. D. Athey, and B. Noble. The end-to-end performance effects of parallel tcp sockets on a lossy wide-area network. In *Proceedings 16th International Parallel and Distributed Processing Symposium*, pages 10–pp. IEEE, 2001.

[280] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: Meeting deadlines in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 41(4):50–61, 2011.

[281] J. Nair, A. Wierman, and B. Zwart. Tail-robust scheduling via limited processor sharing. *Performance Evaluation*, 67(11):978–995, 2010.

[282] K. Walsh and E. G. Sirer. Experience with an object reputation system for peer-to-peer filesharing. In *NSDI*, volume 6, pages 1–1, 2006.

[283] Android interface definition language, 2019.

[284] grpc. `http://www.grpc.io`, 2019.

[285] Lbph face recognizer. `http://docs.opencv.org/trunk/df/d25/classcv_1_1face_1_1LBPHFaceRecognizer.html`, 2019.

[286] L. Lenc and P. Král. Unconstrained Facial Images: Database for face recognition under real-world conditions. In *14th Mexican International Conference on Artificial Intelligence (MICAI 2015)*, Cuernavaca, Mexico, 25-31 October 2015 2015. Springer.

[287] Photo bank of the czech news agency, 2019.

[288] Openalpr. `http://www.openalpr.com/cloud-stream.html`, 2019.

[289] License plate detection, recognition and automated storage, 2017.

[290] Cmusphinx project. `https://cmusphinx.github.io`, 2017.

[291] Tesseract open source ocr engine. `https://github.com/tesseract-ocr/tesseract`, 2019.

[292] J. Jung, S. Lee, M. S. Cho, and J. H. Kim. Touch tt: Scene text extractor using touchscreen interface. *ETRI Journal*, 33(1):78–88, 2011.

[293] Speedtest market report in usa, 2017.

[294] C.-Y. Hong, M. Caesar, and P. Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 127–138. ACM, 2012.

[295] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 435–446. ACM, 2013.

[296] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 1–14. ACM, 2010.

[297] Z. Luo, S. Fu, M. Theis, S. Hasan, S. Ratnasamy, and S. Shenker. Democratizing cellular access with cellbricks. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 626–640, 2021.

[298] M. Johnson, S. Sevilla, E. Jang, and K. Heimerl. dlte: Building a more wifi-like cellular network:(instead of the other way around). In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, pages 8–14, 2018.

[299] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[300] R. Shea, D. Fu, A. Sun, C. Cai, X. Ma, X. Fan, W. Gong, and J. Liu. Location-based augmented reality with pervasive smartphone sensors: Inside and beyond pokemon go! *IEEE Access*, 5:9619–9631, 2017.

[301] Ingress mobile game. `https://www.ingress.com`.

[302] O. C. A. W. Group et al. Openfog architecture overview. *White Paper, February*, 2016.

[303] S. Van Canneyt, M. Bron, A. Haines, and M. Lalmas. Describing patterns and disruptions in large scale mobile app usage data. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pages 1579–1584. International World Wide Web Conferences Steering Committee, 2017.

[304] C. Shepard, A. Rahmati, C. Tossell, L. Zhong, and P. Kortum. Livelab: measuring wireless networks and smartphone users in the field. *ACM SIGMETRICS Performance Evaluation Review*, 38(3):15–20, 2011.

[305] Roundup of internet of things forecasts and market estimates, 2016. `https://www.forbes.com/sites/louiscolumbus/2016/11/27/roundup-of-internet-of-things-forecasts-and-market-estimates-2016/#494bb419292d`.

[306] P. Mell, T. Grance, et al. The nist definition of cloud computing. 2011.

[307] M. Jelasity and O. Babaoglu. T-man: Gossip-based overlay topology management. In *International Workshop on Engineering Self-Organising Applications*, pages 1–15. Springer, 2005.

[308] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1190–1199. IEEE, 2002.

[309] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys & Tutorials*, 7(2):72–93, 2005.

[310] M. Jelasity, A. Montresor, and O. Babaoglu. T-man: Gossip-based fast overlay topology construction. *Computer networks*, 53(13):2321–2339, 2009.

[311] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konečnỳ, S. Mazzocchi, H. B. McMahan, et al. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.