ENABLING AND OPTIMIZING SERVICES WITH EDGE COMPUTING

by

Peng Liu

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2019

Date of final oral examination: 06/24/19

The dissertation is approved by the following members of the Final Oral Committee:

Suman Banerjee, Professor, Computer Sciences, UW-Madison
Aws Albarghouthi, Assistant Professor, Computer Sciences, UW-Madison
Shivaram Venkataraman, Assistant Professor, Computer Sciences, UW-Madison
Younghyun Kim, Assistant Professor, Electrical and Computer Engineering, UW-Madison

To my wife Wenping and daughter Jessie.

Acknowledgments

Seven years ago, I made an important decision to work on the Ph.D. program at the University of Wisconsin-Madison. That was a hard decision because I had been working in the industry for six years and I did not have much research experience. In retrospect, the decision was the greatest one I have made in my life. I am lucky to have the chance to study and conduct interesting research in a nice city — Madison, WI, at a renowned university, with many talented people.

I owe a debt of gratitude to many people. First of all, I would like to thank my advisor, Professor Suman Banerjee. He gave me the opportunity to work in his laboratory. Even though I did not have much research experience before I started the Ph.D. program, he allowed me to explore freely and learn from mistakes. My early stage of Ph.D. was largely driven by curiosity, and I did not always follow his suggestions, so I appreciate his tolerance, patience, and trust. When I realized that I need his help in figuring out research topics and directions or choosing methods, his guidance and support were always invaluable. During my Ph.D., he not only devoted his time and efforts to teach me how to conduct research, write papers, and give presentations, but also helped me a lot on my personal matters.

I also thank Professor Aws Albarghouthi, Professor Shivaram Venkataraman, and Professor Younghyun Kim for serving on my graduate committee. Professor Aws Albarghouthi and Professor Xinyu Zhang served on my preliminary examination committee, I appreciate their time and efforts. Their feedbacks help me shape this dissertation.

And I am fortunate to work with many talented and warmhearted people during

my Ph.D. For example, Doctor Milind Buddhikot taught me how to do research and present ideas during my summer internship at Bell Labs.

I am very grateful for the help and support from senior students in professor Banerjee's group, including Sayandeep Sen, Jongwon Yoon, Tan Zhang, and Ashish Patro. Jongwon and Tan guided me in the early stage of my Ph.D., and they helped me to build the basic skills to do research and write papers. The projects covered in this thesis are joint work with many students and research staffs. Jongwon Yoon, Lance Johnson, and Derek Meyer contributed to the development and deployment of the VideoCoreCluster project. Lance Hartung, Derek Meyer, and Varun Chandrasekaran contributed to the system design of the DeepRTC project. Bozhao Qi helped me on the system design and evaluation of the EdgeEye project. Dale Willis, Arkodeb Dasgupta, Lance Hartung, Derek Meyer, Michael Bauer, Mickey Barboi, Sejal Chauhan, Dongjin Suh, and many other students contributed to the ParaDrop design, development, and deployment. And some of them also contributed a lot to ParaDrop workshops and tutorials. I cannot thank them enough for their help in these projects.

During the seven years, I enjoyed the time working with other students, including Lei Kang, Joshua Tabor, Wei Zhao, Neil Klingensmith, Chuhan Gao, Yilong Li, Yijing Zeng, and many others. Many friends also helped me a lot, such as Liang Wang, Wentao Wu, Junming Xu, Scott Alfeld, Adam Everspaugh, Aaron Cahn, Keqiang He, Yanfang Le, Junaid Khalid, David Tran-Lam, Xiaozhu Meng, Heming Shou, and Yongnam Kim. Even though we did not work in the same area, the discussions with them always gave me some insights. And they also helped me a lot with my personal affairs.

Special thanks to my friend Deng Liu. His encouragement was one of the most important factors that I applied to the Ph.D. program at the University of Wisconsin-Madison and did not give up during my Ph.D.

Finally, thank my family for their support during the seven years. For me, the Ph.D. is a kind of adventure with a lot of challenges. I could not complete it without their continuous support.

Contents

List of Tables

List of Figures

Abstract

In the cloud computing era, a growing trend is to move computing and storage resources from client devices to data centers to achieve high scalability, efficiency, and reliability. However, edge computing, which deploys resources and services close to end-users, has some intrinsic advantages over cloud computing, such as high privacy, low latency, and reduced network traffic. To come up with the best practices to leverage the advantages of edge computing and provide practical gain for both service providers and users, we need to understand what applications can really take advantage of the edge computing paradigm, how to effectively manage the distributed resources at the network edge, and how to seamlessly integrate resources in edge and off-site data centers to provide an easy-to-use API for developers.

Motivated by these questions, we have conducted research on edge computing in three directions: application, framework, and infrastructure. First, we built *VideoCoreCluster*— a highly efficient video transcoder cluster at the edge to enable adaptive live video streaming services. Second, we developed *DeepRTC*— a deep learning based system to optimize real-time video communications at the edge. Third, we implemented *EdgeEye*— an edge application framework providing low latency computation offloading at the edge for computer vision applications. *EdgeEye* encapsulates the heterogeneous hardware for computer vision algorithm accelerations and provides easy-to-use interfaces for developers to build computer vision applications on low-cost devices. Finally, we designed and implemented an open-source edge computing platform — *ParaDrop*, with resources at the "extreme"

wireless network edge, Wi-Fi Access Points (APs). *ParaDrop* focuses on specific design issues around how to structure an architecture, a programming interface, and an orchestration framework through which such edge computing services can be dynamically created, installed, and revoked.

Our work on edge computing application and framework demonstrates the advantages of edge computing for applications requiring high network bandwidth, low latency services, and high privacy. Our work on *ParaDrop* gives the community access to an open-source edge computing platform for research and education purposes, and our experiences uncover some intrinsic complexity to build such a platform for edge computing.

1 Introduction

1.1 Overview

Cloud computing platforms, such as Amazon Elastic Compute Cloud (Amazon EC2), Microsoft Azure, and Google Cloud Platform, have become a popular approach to providing ubiquitous access to services across different user devices. Developers have come to rely on cloud computing platforms to provide high-quality services to their end-users since they are reliable, always on, and robust. Netflix and Slack are examples of popular cloud-based services. Cloud services require developers to host services, applications, and data on offsite datacenters. That means the computing, networking, and storage resources are spatially distant from end-users' devices. However, a growing number of high-quality services desire computational tasks to be located nearby. These services include the need for lower latency, greater responsiveness, a better end-user experience, and more efficient use of network bandwidth. As a consequence, over the last decade, a number of research efforts have studied the need and benefits of creating edge computing services that distribute computational functions closer to the client devices, e.g., Cyber Foraging [1], Cloudlets [2], and more recently Fog Computing [3]. The industry is currently standardizing Multi-access Edge Computing (MEC) [4], previously known as Mobile Edge Computing [5]. MEC provides computational and storage

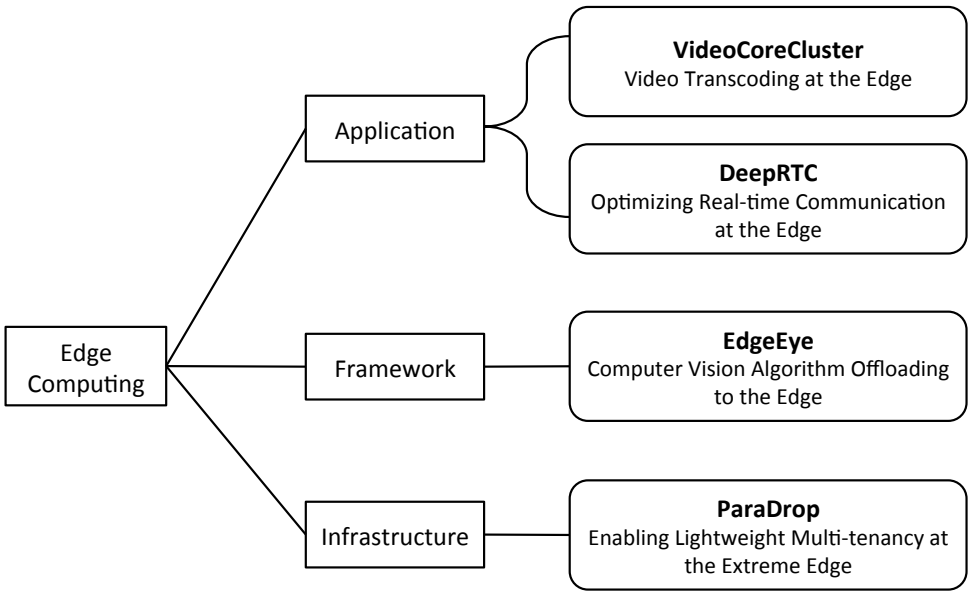


Figure 1.1: Our work on edge computing. We completed two pieces of work on edge computing applications — enabling a live video service with edge computing (*VideoCoreCluster*) and optimizing real-time video communication with edge computing (*DeepRTC*). And we built a computer vision algorithm offloading framework for edge applications (*EdgeEye*). In addition, we have been working on an open source edge computing platform (*ParaDrop*).

resources at the access network in order to reduce latency, ensure highly efficient network operation and improve user experience.

In this dissertation, we try to understand the advantages of edge computing and how to leverage those advantages in practical applications. We aim to answer the following questions: *(a) What types of applications can benefit from edge computing over cloud computing? (b) Can we build application frameworks to simplify the edge application development? (c) How can we build an efficient and flexible edge computing platform to manage resources at the network edge and provide convenient interfaces to users, developers, and resource owners in order to maximize the benefits of edge computing? (d) How can we harmoniously integrate cloud and edge computing to transparently optimize the user experience?*

To answer these questions, we have been working on edge computing in three directions — application, framework, and infrastructure. An overview of our methodology on edge computing research is contained in Fig. 1.1. We studied two applications to understand the benefits that we can get by using the edge-computing paradigm(*VideoCoreCluster* [6] and *DeepRTC*). We built an edge application framework — *EdgeEye* [7], for computer vision algorithm offloading. Developers can use the framework to build Internet of Things (IoT) applications or mobile Augmented Reality (AR) applications. In addition, we have been building an edge computing platform — *ParaDrop* [8, 9], to implement a framework to manage resources and services at the extreme network edge. Building such a system is a great opportunity to understand the requirements of edge computing applications and the source of the advantages. We also have the chance to understand the complexity to manage the distributed resources and services. Moreover, the availability of that platform provides a convenient base for us to easily try edge computing applications in the wild at a large scale.

In our work, we focus on the following three categories of edge computing applications:

- High bandwidth content delivery, such as video streaming and real-time video communications (in *VideoCoreCluster* and *DeepRTC*).
- Computation offloading, e.g., computer vision algorithms and mobile Augmented Reality (AR) (in *EdgeEye*).
- Sensor data aggregation and device control in IoT applications (in *ParaDrop*).

1.2 Background

Video applications, especially the ones with requirements for low latency or high privacy, can take advantages of edge computing. Effective video compression is crucial to guarantee good quality of service by reducing the amount of data needed to represent video contents, and video transcoding (re-encode compressed video

to multiple versions) is necessary to provide adaptive bitrate streaming support — which is helpful to support mobile devices with dynamic network performance. In addition, video content analysis can be used to build many interesting applications. However, those operations, such as video transcoding and content analysis, are computationally expensive, especially for mobile devices. Also, even with compression, video data needs high bandwidth to transmit, and in many cases, users have concerns about privacy, so offloading the tasks to the cloud is not an option. Considering the requirements for low latency, reduced bandwidth usage, and high privacy, edge computing is suitable for many video applications. In this dissertation, we explore the applications of edge computing on video transmission and content understanding.

Our work on edge computing applications demonstrate the advantages of the edge computing paradigm. However, we believe the adoption of edge computing depends on the availability of the easy-to-use tools and infrastructures to develop and deploy edge computing applications. Edge computing is a blurred concept. In the path that a packet travels from end devices to servers in a datacenter, multiple places can be considered as the edge, such as home network, enterprise network, and access network. We chose to build an edge computing platform for the extreme edge — WiFi routers. We call the platform *ParaDrop*, because it can "paradrop" services from the cloud to the edge.

More details of our work in the three directions are as follows:

1.3 Optimizing Live Video Streaming and Real-time Video Communications with Edge Computing

First, we used the edge computing approach to optimize video applications. Our research focused on two specific video applications requiring low latency — live video streaming and real-time video communications. Video data in these use cases have the following characteristics which make them suitable for edge computing:

- Videos require high bandwidth, so uploading them to the cloud continuously

can introduce network congestion, which can be avoided by processing the data locally at the edge.

- Live videos need to be processed with low latency and archiving is not required.
- Video data processing requires high computational resources and in many cases, hardware accelerators are required.

In the live video streaming project, we enabled and optimized a live video streaming service with edge computing. In order to support mobile devices with diverse browsers and operating systems, video transcoding was required to build the service. By using transcoders at the network edge, we reduced the bandwidth usage as well as the corresponding cost by eliminating the requirement to upload live video data to cloud consistently. In order to further optimize the user experience, we built a low-cost video transcoder cluster — *VideoCoreCluster*, to transcode source videos to different bitrates and qualities, so that mobile devices (players) can select the best bitrate to play according to their wireless network performance. We implemented the video transcoder cluster with specialized hardware video decoder and encoder to achieve high power efficiency, and employed a scheduling algorithm to make sure the cluster provides a reliable transcoding service to the live video streaming service.

In the real-time video communication project, we wish to understand whether *insight about the content being compressed can be used to optimize video transmission*. We designed and implemented *DeepRTC* which provides two optimizations atop vanilla compression schemes for real-time communications: (a) sender-side content-aware encoding, and (b) receiver-side quality enhancement through information extraction and infusion. The content-aware encoding scheme utilizes the Deep Neural Network (DNN) based object detection and semantic segmentation techniques to hierarchically classify pixels for bit allocation. Pixels of a user's interest are encoded with more bits (and consequently higher quality), while less important pixels receive fewer bits in encoding. The quality enhancement scheme employs

content and quality-aware DNN-based super-resolution techniques to enhance the perceived quality of the decoded frames. Essentially, the quality enhancement scheme uses DNN models to transmit information (or prior knowledge for the receiver). The DNN models are trained in the sender side to extract information from representative video frames, and in the receiver side, the trained DNN models infuse the extracted information back into the video frames through the DNN inference. The trained DNN models can be transmitted with a side channel, and they can be reused if possible. Our prototype implementation is built upon the VP9 codec and WebRTC. Based on our evaluations, we observed that the first optimization can reduce the bitrate by 50%, or improve the quality of important pixels by 2dB in terms of SSIM and PSNR. The second optimization can improve the video quality up to almost 2dB in terms of SSIM and up to 5dB in terms of PSNR. Both the two optimization schemes require high computational resources. And hardware accelerator is required for real-time execution of them. Current mobile devices do not have enough resource for them. In addition, cloud computing cannot help much, because the network path between the end device and cloud server may be a bottleneck for high bandwidth video data. Privacy is another concern for the cloud-based approach because real-time video data might be sensitive. So we had to leverage the edge computing scheme to build the system.

1.4 A Low-latency Computer Vision Algorithm Offloading Framework at the Edge

With the rapid growing capabilities of machine learning techniques, more and more developers are integrating machine learning algorithms into their applications to provide new functions or improve the currently available features to end users. However, machine learning techniques, especially deep learning, face some challenges to be deployed in end devices, including the lack of high-performance hardware accelerators and diverse programming environments [10]. An application framework providing task-specific interfaces to developers by hiding the diversity

of hardware accelerators and programming environments will be very useful for many application developers. Based on this observation, we worked on such a framework for applications.

We chose to focus on computer vision algorithms in this piece of work. Camera sensors are available in many low-cost devices. Properly using the camera sensor data (image/video) can provide interesting and useful features to users. Decades of research has generated many computer vision algorithms to analyze the image and video data. Among these algorithms, deep learning based approaches can achieve state-of-the-art accuracy on many computer vision tasks. However, because of the high demand for both computation and storage resources, DNNs are often deployed in the cloud instead of at the end devices. Unfortunately, executing deep learning inference in the cloud, especially for real-time video analysis, often incurs high bandwidth consumption, high latency, reliability issues, and privacy concerns. Moving the DNNs close to the data source with an edge computing paradigm is a good approach to address those problems. Moreover, the lack of an open source framework with a high-level API also complicates the deployment of deep learning-enabled service at the Internet edge.

We built *EdgeEye*, an edge-computing framework for real-time intelligent video analytics applications. *EdgeEye* provides a high-level, task-specific API for developers so that they can focus solely on application logic. *EdgeEye* does so by enabling developers to transform models trained with popular deep learning frameworks to deployable components with minimal effort. It leverages the optimized inference engines from industry to achieve the optimized inference performance and efficiency.

1.5 An Edge Computing Platform at the Extreme Edge

With several pieces of work being done in the applications of edge computing, we turned our focus to the edge computing infrastructure. We have two goals on the

edge computing infrastructure project: i) understanding the complexity to build a useful platform for both developers and users; and ii) providing such a platform so that developers can easily build and deploy edge services for end users.

In this work, we built an edge computing platform based on Wi-Fi routers. A Wi-Fi router is a ubiquitous device in home and enterprise. It is always on and connected. Moreover, as the improvement in hardware capabilities, it has enough performance and resources to run many edge computing applications. Thus, it is an ideal place to deploy edge computing services. We name the platform *ParaDrop*, and a service running on the platform a *chute* (short of parachute). The goal of *ParaDrop* is making the job of "dropping" service from the cloud to the edge easily. We implemented the system with high efficiency and flexibility, and we defined an easy to use API for users and third-party developers. We built *ParaDrop* based on a light-weight virtualization technique — Docker [11], to provide an isolated environment for third-party services sharing the same hardware resources on the router. *ParaDrop* also support cloud-edge hybrid applications, which means developers can divide a service into multiple pieces, and deploy these pieces on edge (*ParaDrop*) and cloud accordingly. In the process of developing *ParaDrop*, we implemented two IoT applications — *Environment Sense* (*EnvSense*) and *Security Camera* (*SecCam*). Those applications are typical examples of IoT data aggregation and control. They demonstrate two advantages of edge computing: high privacy and reduced network traffic.

1.6 Contributions

We built practical systems to understand the advantages of using edge computing paradigm and the complexities of building an edge computing infrastructure. This dissertation describes the design and implementation of the systems, as well as the evaluation results.

In this dissertation, we focus on video applications, such as video transmission and intelligent video analytics. Those applications demonstrate the advantages of edge computing paradigm on low latency and reducing network bandwidth, and

since the data is stored and processed locally, edge computing also helps to protect users' privacy.

On edge computing infrastructure, we chose to build a specific edge computing platform using WiFi routers as the edge nodes. A WiFi router is one of the possible locations that can be considered as the "edge". We built it primarily for research and educational purposes. With the understanding of the benefits of the edge computing paradigm, the availability of an easy-to-use edge computing infrastructure will definitely speed up the research on edge computing applications and also the deployment of practical systems leveraging the power of edge computing.

Specifically, our contributions of this dissertation are the following:

1. The work of *VideoCoreCluster*. Through video transcoding at the edge network (campus network), we enabled and optimized a live video streaming service for users with diverse devices. We built a cost-effective and power-efficient transcoding solution based on hardware video decoding and encoding modules. We leveraged characteristics of adaptive bitrate video streaming over HTTP to design a reliable and scalable system. The system is easily deployable. We employed the *VideoCoreCluster* into an IP-based TV streaming service at the University of Wisconsin-Madison.

2. The work of *DeepRTC*. We propose *DeepRTC* which includes two optimization approaches to enhance current video transmission systems for RTC applications. *DeepRTC* analyzes video content and allocates more bits for more important pixels at the sender, and it leverages a lightweight information extraction and infusing approach to enhance the perceived video quality at the receiver. We evaluated the gains of *DeepRTC* with benchmarks in terms of bandwidth reduction, quality improvement, and speed. We implemented a prototype of *DeepRTC* based on PyTorch, TensorFlow, VP9 codec, WebRTC, and GStreamer. With the edge computing paradigm, *DeepRTC* can improve the performance of RTC applications on mobile devices in terms of bandwidth usage and video quality.

3. The work of *EdgeEye*. We propose an edge service framework for real-time video analytics applications — *EdgeEye*, which provides a high-level abstraction of some important video analysis functions based on DNNs. Applications can easily offload the live video analytics tasks to the *EdgeEye* server using its API, instead of using deep learning framework specific APIs. We built a prototype of *EdgeEye* with Caffe and Nvidia's TensorRT. Our preliminary evaluations indicate that *EdgeEye* provides higher inference performance than using the deep learning frameworks directly. We present an example application built with *EdgeEye* to show the capabilities of the framework.

4. The work of *ParaDrop*. We observed the unique challenges in a Wi-Fi AP based edge computing platform with virtualization techniques and proposed corresponding solutions to overcome these challenges. We designed the system architecture and fully implemented it on hardware to provide a reliable and easy to use edge computing platform for developers to deploy edge computing applications. In this dissertation, we introduce the process of developing applications for *ParaDrop* with two example applications. And we also discuss other possible applications for this platform. In addition, we analyze and evaluate the system in terms of efficiency and effectiveness, and discuss its flexibility and scalability.

1.7 Outline

The remaining dissertation is organized as follows: First, we introduce our work on two edge computing applications in Chapter 2 and Chapter 3. Second, Chapter 4 presents the computer vision algorithm offloading framework. After that, we introduce our work on the edge computing platform — *ParaDrop*, in Chapter 5. In each chapter, we introduce the necessary background, followed by the details of our work and evaluation results. We discuss the related work in Chapter 6, and conclude this dissertation and discuss future research directions in Chapter 7.

2 Low-cost Video Transcoding at the Edge

2.1 Introduction

In this chapter, we introduce our work on designing and deploying a video transcoder cluster at the Internet edge (campus network). Through video transcoding, we enabled a live video streaming service to support mobile devices with diverse operating systems and web browsers.

Video streaming service is one of the most popular Internet services in recent years. In particular, multimedia usage over HTTP accounts for an increasing portion of today's Internet traffic [12]. In order to provide high and robust quality of video streaming services to end users with various devices with diverse network connectivities, content owners and distributors have to encode the video to different formats, bitrates, and qualities. The broad spectrum of varieties of bitrates, codecs, and formats make it difficult for some video service providers to prepare all media content in advance. And it is redundant to encode and store a source video to different variants for archiving purposes. Therefore, video transcoding has been widely used for optimizing video streaming services. For example, Netflix encodes a movie as many as 120 times before they stream the video to users [13]. Transcoders are also commonly used in the area of mobile device content adaptation, where a target device does not support the format or has limited storage capacity and

computational resource that mandate a reduced file size.

However, video transcoding is a very expensive process, requiring high computational power and resources. In addition, video transcoders' performance is important to enhance the overall quality of a video streaming service. Regarding the performance of a transcoder, two metrics are important: quality and speed. Video quality with a given bitrate determines the amount of data needed to be transmitted in the network for a video. Transcoding speed determines the time to complete the transcoding. Thus, it is critical for live video streaming services because the transcoding must be done in real-time. Other metrics, e.g., cost and power consumption, also need to be considered in a video transcoding system deployment.

Various video transcoding technologies are proposed and used, including cloud transcoding, software-based transcoding on local servers, and specialized hardware-based transcoding. In this chapter, we introduce *VideoCoreCluster*–a low-cost and energy-efficient hardware-assisted video transcoder cluster to provide transcoding services for a live video streaming service. The cluster is composed of a manager and a number of cheap single board computers (Raspberry Pi Model B). We use the hardware video decoder and encoder modules embedded in the System on Chip (SoC) of the Raspberry Pi to facilitate video transcoding. With an optimized transcoding software implementation on the Raspberry Pi, each Raspberry Pi is able to transcode up to three standard definition (SD, 720x480) videos or one high definition (HD, 1280x720) video along with one SD video in real-time with very low power consumption. We developed the cluster manager based on an IoT machine-to-machine protocol - MQTT [14] to coordinate the transcoding tasks and hardware transcoders in order to provide reliable transcoding service. Compared to software-based video transcoders, *VideoCoreCluster* has lower cost and higher energy efficiency.

Adaptive bitrate (ABR) video streaming over HTTP is a popular technology to provide robust quality of service to end users whose mobile devices have dynamic network connectivities. Multiple ABR technologies are used in the industry, but they share a similar idea. Media servers split source videos into small segments and

encode every segment to multiple variants with different bitrates. During playback, a video player selects the best variants of these segments base on the video player's performance, network connectivity, or user's preference. Streaming video over HTTP has many advantages [15]. We chose it for our live video streaming service because it is easy to deploy a video player on web browsers of mobile devices with different operating systems, and we do not need to reconfigure middle-boxes (firewalls, NATs, etc.) in the campus network for our service.

2.2 Background

We worked with the IT department of the University of Wisconsin-Madison to provide a free TV service for students on campus. Students could watch 27 (as of 2016) TV channels with their mobile devices. Six of these channels are HD channels with a resolution 1280x720, 30fps. The other 21 channels are SD with resolutions up to 720x480, 30fps. In the month of April 2016, there were more than 4000 view sessions and about a total of 480 watching hours.

ABR video streaming techniques are used to provide high-quality video streaming service to mobile devices with diverse link capabilities and network dynamics. The system supports both Apple's HTTP Live Streaming (HLS) [16] and MPEG-DASH [15] to provide service to heterogeneous devices. Fig. 2.1 shows the architecture of the system. Specifically, a TV frontend receives TV signal and encodes the video stream into H.264 + AAC format, then the video stream is pushed to the *source video server*. The *transcoder cluster* pulls videos from the *source video server* and pushes the transcoded results to the *media server* in order to provide multiple variants of the video streams for every TV channel. The *source video server* and *media server* can be combined in deployment, so we will not discuss them separately in the following sections. The web server hosts web pages and video players for different web browsers.

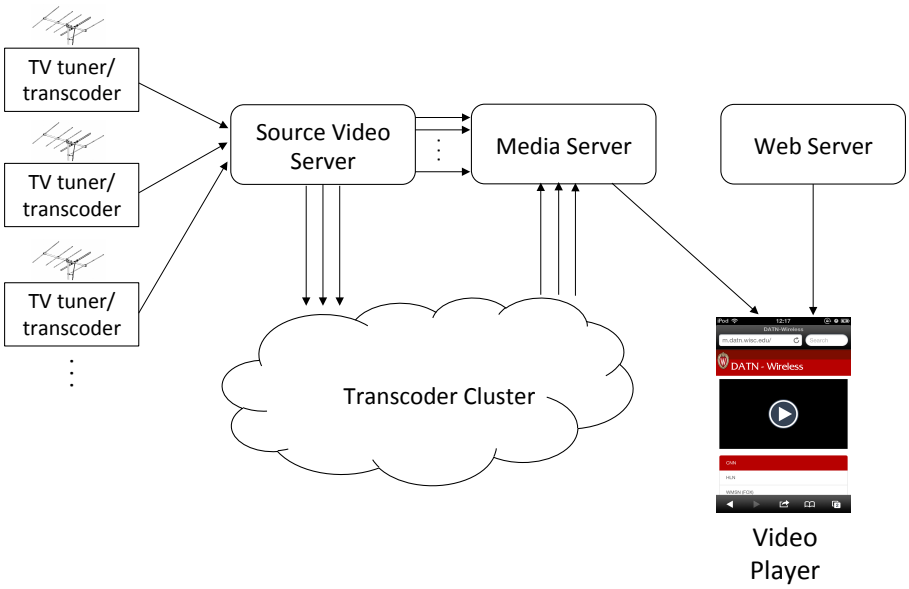


Figure 2.1: TV streaming service architecture.

Challenges of ABR Techniques on Live Video Streaming

Low latency is critical for a live video streaming service, in which a media server has to generate video data on-the-fly to provide continuous streaming service to end users. A media server supporting ABR needs to guarantee all variants of a video segment are ready when a client requests one of them. Due to the strict requirement on low-latency transcoding, various optimization techniques (e.g., high throughput video transcoding by parallelism and multi-pass encoding to enhancing video quality) are not applicable for live streaming. Moreover, an index file containing the list of available video variants need to be generated in real-time, and it has to be updated on time and be consistent with the availability of these variants. We also need to make sure the variants of a video segment are generated synchronously to simplify the implementation of ABR algorithms on the video players. Furthermore, an efficient video transcoding system is desired and high reliability is required to guarantee 24/7 video streaming services to users without interruptions.

Video Transcoding

Essentially, a video transcoder is composed of a video decoder and an encoder. Some researchers have studied optimized video transcoder designs, which mingle the modules of a video decoder and an encoder [17]. However, having separate video decoders and encoders provides more flexibility because we can easily have various decoder/encoder combinations to create different video transcoders for different purposes. In this work, we only discuss the video transcoder built by a separate video decoder and an encoder.

Most popular video codecs have well-defined standards. These standards strictly define the compliant bitstreams and decoder's behaviors. For many video coding standards, including H.264 [18], different video decoder implementations are required to generate the identical video frames (output) with respect to the same input. This makes the selection of video decoder straightforward, and hence, due to its high efficiency, hardware video decoders would be the best option in most cases as long as it is available at low cost.

On the other hand, developers are free to design a video encoder's implementation as long as the generated bitstream can be decoded by the reference decoder implementation. Therefore, different encoder implementations can generate different bitstreams with different qualities for the same video frames. In this way, the interoperability is guaranteed while innovations on the encoder design are encouraged. As a result, we need to evaluate an encoder's performance on video quality in addition to the encoding speed when we select an encoder for a transcoder. Software video encoders are well known for their low efficiency. Hameed et al. [19] pointed out that application-specific integrated circuit (ASIC) implementation of a video encoder is 500 times more energy efficient than the software video encoder running on general purpose processors. They assume that both hardware and software implementations use the same algorithms for the corresponding procedures, e.g., motion estimation, intra-prediction, etc.. The high efficiency is only from the advantage of specialized hardware implementation. However, software video encoders are much more flexible than hardware video encoders. It is much easier

to try new algorithms on a software video encoder to improve video quality than a hardware video encoder. FPGA-based video encoders achieve a good balance between efficiency and flexibility, so they are also widely used in the industry.

Different video applications have different requirements on video encoder. For mobile devices, energy efficiency is crucial for battery life. Therefore, most of the SoCs for mobile phones include hardware video decoder/encoder IP (intellectual property) cores [20, 21]. For broadcast applications, high video quality is desired while real-time encoding and flexibility are critical. Toward this, video encoders on the FPGA platform are widely used. For on-demand streaming applications, the low energy efficiency of software video encoders can be amortized by streaming one encoded video to many users. High latency is not a big concern because the service provider can encode video offline. Slow encoding speed can be overcome by launching a large number of instances in a cloud platform to run many video encoders in parallel to achieve very high throughput video encoding. The advantages of software encoders on video quality and flexibility are the main reason that they are widely used to prepare video contents for on-demand streaming services. For instance, Netflix adopted the software-based transcoder because of its flexibility, after an unsuccessful deployment of a specialized hardware video transcoding system [22].

Since H.264 is the video coding standard that our system supports, we only discuss the available H.264 encoder implementations in this chapter as shown in Table 2.1. The authors of [32] compared different H.264 encoder implementations, which includes software implementations (x264, DivX H.264, etc.), GPU-accelerated implementation (MainConcept CUDA), and hardware implementation (Intel QuickSync Video). Their conclusions include *(i)* x264 is one of the best codecs regarding video quality, and *(ii)* Intel QuickSync is the fastest encoder of those considered. Even though x264 is the best software H.264 encoder option for our project, its low efficiency hinders its deployment. Given that we need to keep the system running 24/7 for continuous TV service in the campus network, it is not economical to rent cloud computing resource (e.g., Amazon EC2 instances) to execute the transcoding tasks. Building an in-house transcoding system with highly

Encoder Type	Explanations
Software	JM: The reference H.264 implementation from JVT [23]. It is widely used for research purpose and conformance test. It is too slow to be used in practical projects. x264: The most popular open source software H.264 encoder implementation. Compared to JM, it is about 50 times faster and it provides bitrates within 5% of the JM reference encoder for the same PSNR [24, 25]. OpenH264: An open source H.264 implementation from Cisco [26]. It is optimized and the encoder runs much faster than JM, but it is slower and has fewer features than x264. Other proprietary implementations: MainConcept [27], Intel's IPP H.264 encoder [28], etc.
GPU-based	Three GPU vendors: Intel, NVIDIA, and AMD, all integrate hardware video codecs in their GPUs. They also provide GPU-accelerated video encoders, which leverage GPU's high throughput graphics engine to accelerate video encoding [29, 30]. For example, NVIDIA has two different versions of video encoder implementations: NVCUVENC and NVENC. NVCUVENC is a CUDA software-based implementation while NVENC is based on dedicated encoding hardware engine. NVCUVENC will not be available in the future because NVENC provides good performance and quality.
FPGA-based	Xilinx and its partners have professional solutions for broadcast applications [31]. They provide H.264 decoder and encoder IPs for Xilinx's FPGA platforms. FPGA-based implementation is more flexible than ASIC implementation and more efficient than software implementation. But it is more expensive than both of them for small scale deployment.
Encoder IP in SoC	Many SoCs for mobile devices or other embedded devices have dedicated hardware decoders and encoders. Examples include Qualcomm's chips for mobile phones [20] and Broadcom's chip for TV set-top boxes.

Table 2.1: Different type of H.264 encoders

efficient hardware video decoders and encoders is the best choice to implement a low-cost system.

We used the hardware H.264 decoder and encoder in a low-cost SoC — Broadcom BCM2835, which is the chip of a popular low-cost single board computer — Raspberry Pi Model B. *VideoCoreCluster* leverages the SoC's powerful GPU — VideoCore IV, which embeds hardware video decoders and encoders. The excellent software support [33, 34, 35] of the Raspberry Pi was another reason we chose it to build the system.

2.3 *VideoCoreCluster* Architecture

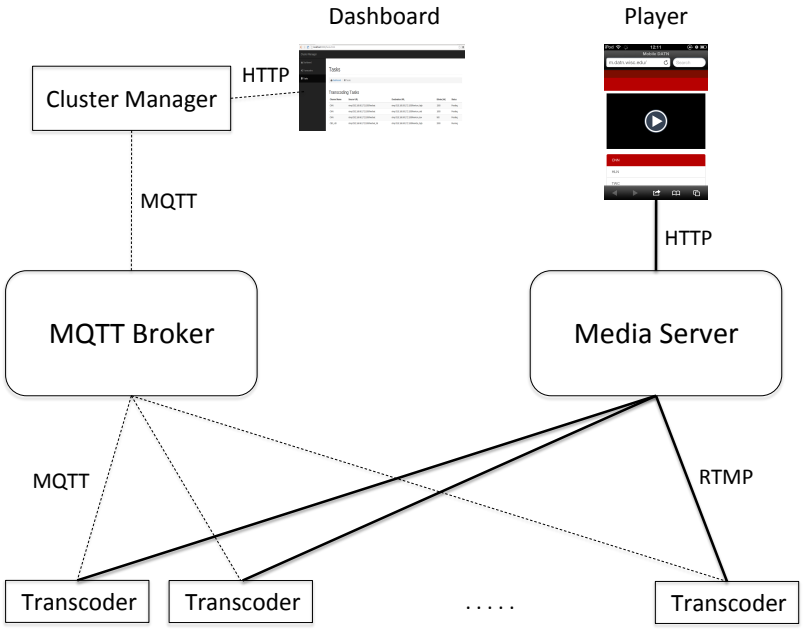


Figure 2.2: The components and connections between them in the *VideoCoreCluster*. Every transcoder node maintains two separate network connections for the control flow and data flow respectively. The administrator monitors the status of *VideoCoreCluster* and updates transcoding tasks through the dashboard of the cluster manager.

The *VideoCoreCluster* is composed of a cluster manager and a number of transcoders. We use the MQTT protocol to transfer control signals between the cluster manager and the transcoders. MQTT is based on a publish/subscribe messaging pattern rather than a traditional client-server model, where a client communicates directly with a server. The cluster manager and the transcoders connect to an MQTT message broker. They exchange information by subscribing to topics and publishing messages to topics. The message payload is encoded with Google Protocol Buffer for minimal overhead [36]. RTMP [37] is used in the data path. Fig. 2.2 shows the architecture of the *VideoCoreCluster*.

Media Server

The media server supports HLS, MPEG-DASH, and RTMP. HLS and DASH are for the video players, whereas RTMP is for the transcoders. HLS and DASH are two popular adaptive bitrate streaming standards to stream video over HTTP and they are widely supported by mobile operating systems and web browsers. RTMP is designed to transmit real-time multimedia data, and thus, it guarantees to transfer source video to the transcoders and the transcoded video to the media server with minimum latency. RTMP's session control features for media applications are used to implement the interactions between the media server and transcoders in our configuration.

Fig. 2.3 shows the responsibilities of the media server in our system. For every TV channel, there are multiple transcoded video streams (variants) with different bitrates and qualities. Each video stream is split to chunks with the same duration on the media server. It is important to align the boundaries of those chunks from different variants even different workers may not be exactly synchronized. To ensure synchronization, we set the Instantaneous Decoder Refresh (IDR) interval of a source video and the transcoded video to 2 seconds. Then the media server uses the IDR frame as the boundary to split the streams. It uses the timestamp of the IDR frame with 2 seconds (IDR interval) granularity to define the segment number. So that we can keep the chunks to be synchronized from a video player's perspective as long as the progress offset of different workers for the same source video is under 2 seconds. We can set the IDR interval to a larger value to obtain higher tolerance on transcoding speed variation. The index file for a channel provides information about all variants of the video. It is dynamically updated by the media server based on the availability of the video streams. One worker may temporarily fail to generate the corresponding stream, and the transcoder cluster can migrate the failed task from one worker to another within a second. RTMP specification requires the initial timestamp of a stream to be 0 [37]. To ensure all other RTMP streams of the same channel as the failed stream have the same timestamp, we need to reset their RTMP connections as well. The media server can always generate a

consistent index file so that we can guarantee the reliability of the video streaming service.

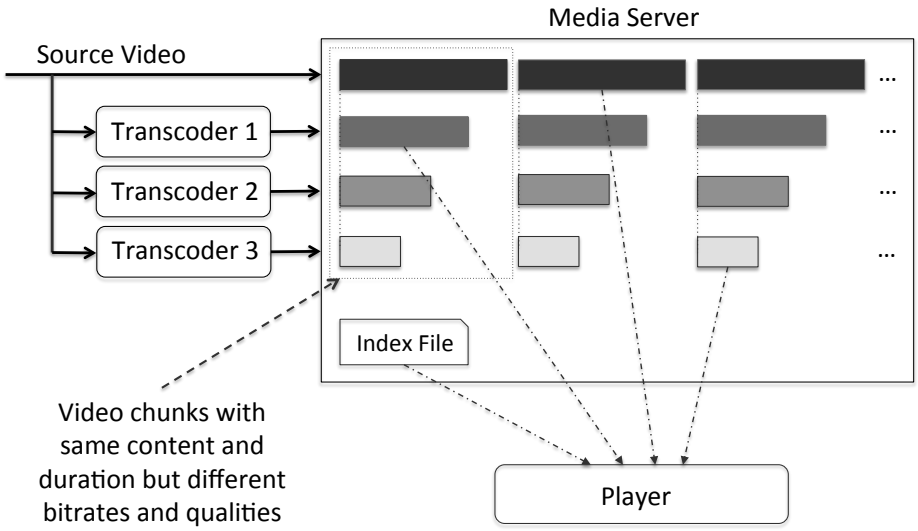


Figure 2.3: Overview of the media server's responsibilities: (i) Splitting video streams to chunks with the same duration (IDR interval). (ii) Refreshing the index file according to the status of the transcoded streams.

Transcoder Design

The Raspberry Pi Model B has a weak ARM CPU but a powerful GPU (VideoCore IV). Given that, our design strategy is to run simple tasks on the CPU and offload compute-intensive tasks to the GPU. There are two types of processes in a transcoder: a cluster agent and zero to three transcoding workers depending on the transcoding task's requirement on the computing resource. Fig. 2.4 shows the relationship between these two types of processes.

The cluster agent executes on the ARM processor only. It has two responsibilities: (i) Reporting the status of the board and workers to the cluster manager. (ii) Accepting commands from the cluster manager and launching or killing transcoding

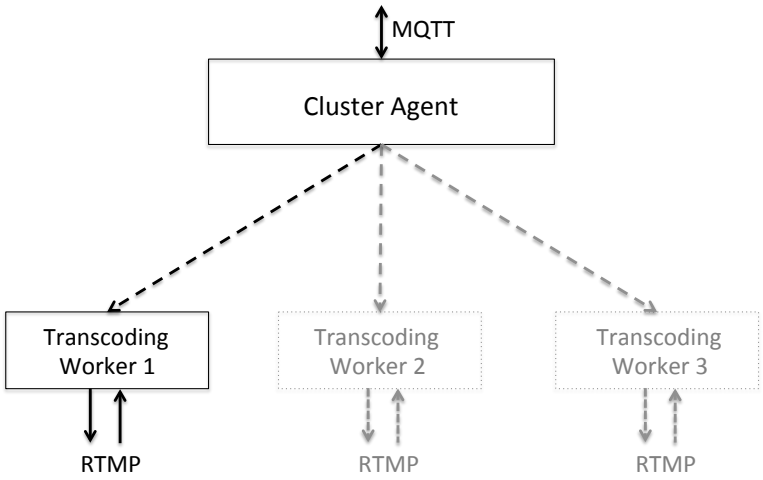


Figure 2.4: Two types of processes on a transcoder. A transcoding worker is the child process of the cluster agent process.

worker processes. A cluster agent will report the available resource to the cluster manager when it registers to the cluster manager. The cluster manager dispatches transcoding tasks to a transcoder based on the available resources of the transcoder. A cluster agent maintains an MQTT connection to the MQTT message broker by sending keep-alive packets if no information flows between the transcoder and the MQTT message broker for a predefined interval. If a cluster agent is disconnected from the broker or the cluster manager is offline, the transcoder will stop all transcoding workers. The cluster manager can revoke a transcoding task from a transcoder if it wants to assign the task to another transcoder. Failed transcoding tasks will be reported to the cluster manager, which can assign them to other transcoders.

A transcoding worker is a process to transcode a video stream in real-time. It only transcodes video data but passes through audio data. Video transcoding tasks primarily execute on the VideoCore IV co-processor. The ARM processor is responsible for executing the networking protocol stack, video streams demuxing and muxing, and audio data pass through. In BCM2835, an application layer software can access the hardware video decoder and encoder through the OpenMAX IL

interfaces [33, 38]. Rather than calling the OpenMAX IL interfaces directly, we built the program with the GStreamer framework [39]. GStreamer is an open source multimedia framework widely used to build media processing applications. It has a well-designed filter (plugin) system. The *gst-omx* is the GStreamer OpenMAX IL wrapper plugin that we used to access the hardware video decoder and encoder resources. Fig. 2.5 shows the software architecture of a transcoding worker.

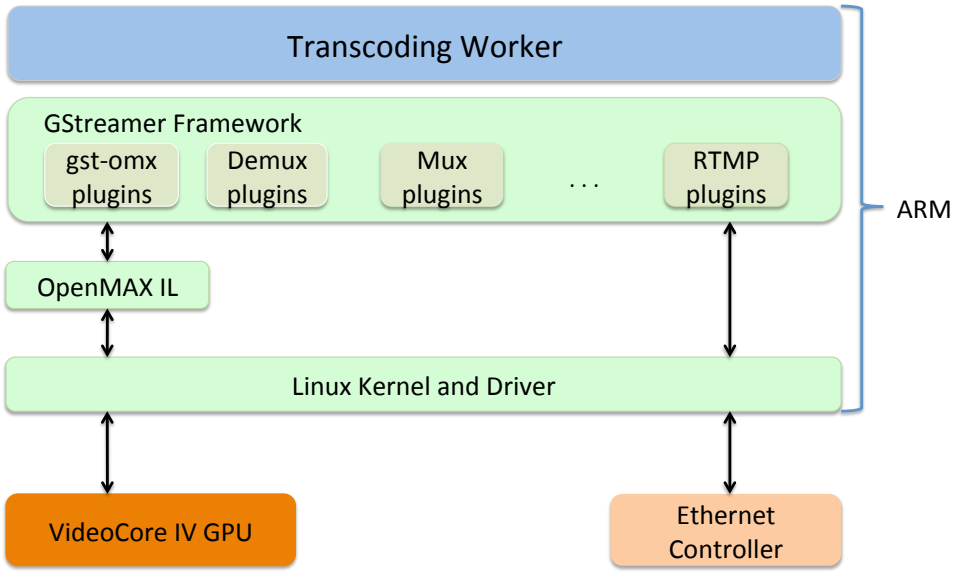


Figure 2.5: The software architecture of a transcoding worker. Compute-intensive video transcoding task executes on the VideoCore IV GPU, whereas the ARM processor is responsible for the coordination and data parsing/movement.

A transcoding worker maintains a pipeline of GStreamer plugins. Video data are processed by the plugins one by one, sequentially. Fig. 2.6 shows the structure of a pipeline. All plugins except H.264 decoder and H.264 encoder plugins fully execute on the ARM processor.

The default behavior of a GStreamer plugin can be summarized as three steps: (i) read data from the sink pad, (ii) process the data, and (iii) write data to the source pad. The GStreamer pipeline moves data and signals between the connected source pad and sink pad. Plugins work independently and process the data se-

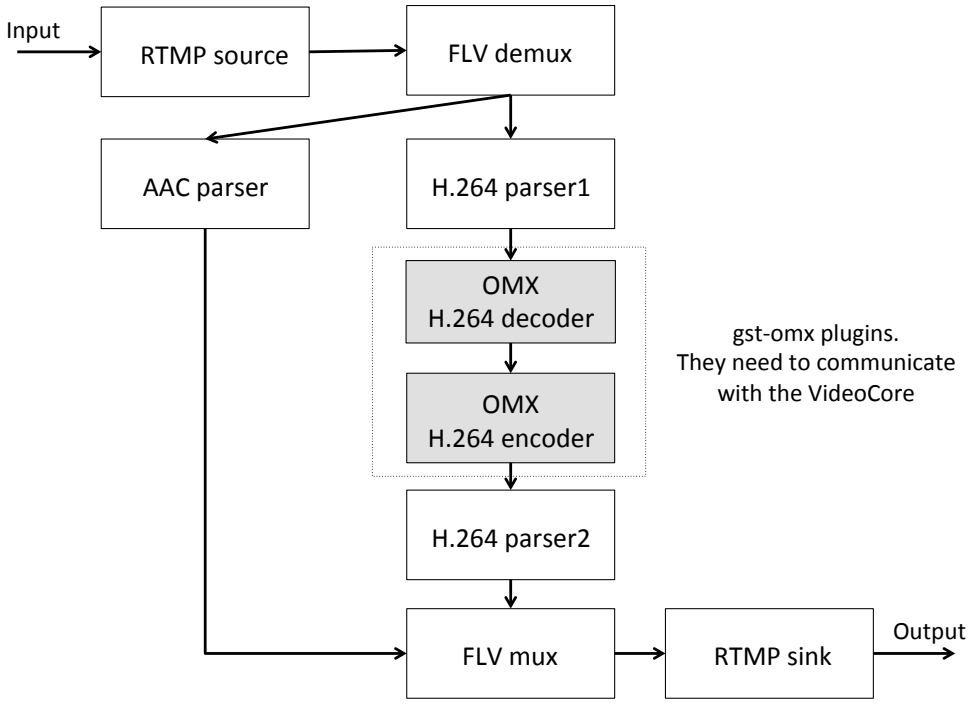


Figure 2.6: The GStreamer pipeline of a transcoding worker. Some trivial plugins, e.g., buffering, synchronization, etc., are not shown here.

quentially. This means that both the H.264 decoder and H.264 encoder need to move a large amount of data back and forth between the memories of the ARM processor and the VideoCore IV GPU, which wastes CPU cycles. We modified the gst-omx implementation to enable hardware tunneling between the decoder and encoder, which significantly reduce the CPU load [40]. Without the hardware tunneling, a transcoder worker cannot transcode a 1280x720, 30fps video in real-time. Whereas after we enable it, a transcoder worker can simultaneously transcode one 1280x720 video and one 720x480 video in real-time. Fig. 2.7 illustrates the data movement in the pipeline without hardware tunneling. And Fig. 2.8 illustrates the data movement in the pipeline with hardware tunneling.

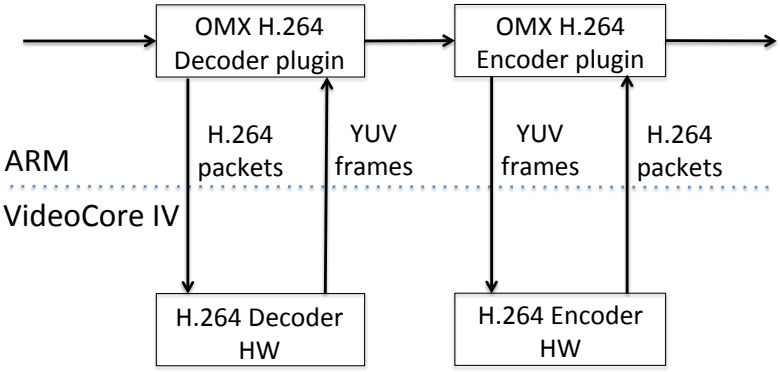


Figure 2.7: The data movement when a decoder and an encoder work independently. YUV frames need to be moved from the VideoCore IV memory to the CPU memory by the decoder plugin, then they need to be moved from the CPU memory to the VideoCore IV memory by the encoder plugin.

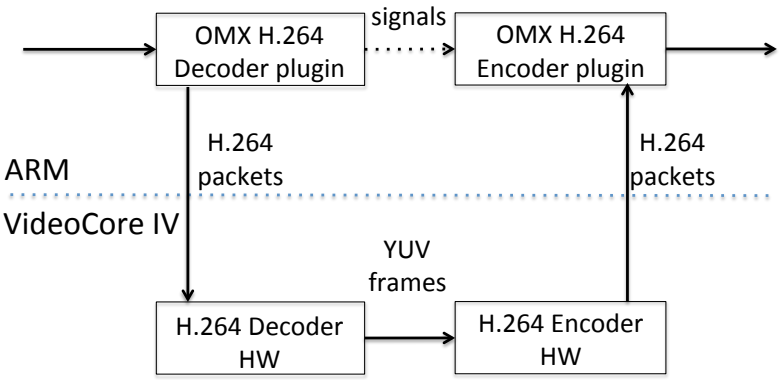


Figure 2.8: A hardware tunnel is created between the decoder and the encoder. Plugins do not need to touch the YUV frames. Therefore, the workload of the ARM processor is reduced.

Cluster Manager Design

The cluster manager maintains a task pool and a transcoder pool. Its major responsibilities are assigning the transcoding tasks to the transcoders, and migrating the failed tasks from one transcoder to another. Both tasks and transcoders have priorities, which need to be considered when the cluster manager schedules the

tasks. The cluster manager maintains a series of lists of tasks with different priorities. Every transcoding task has the following information: { ID, channel name, command, bitrate, resource, priority, state }. The resource is an integer representing the required computational resource for the task. Its value depends on the source video type (SD or HD) and target bitrate. Each task has four possible states: idle, assigning, revoking, and running. The cluster manager employs an event-driven design. When anything happens to the tasks or transcoders, e.g., task failure, a new worker joining, a worker leaving, etc., the cluster manager will check whether rescheduling is necessary and do so if it is.

When a new worker sends a register message to the cluster manager, the cluster manager will add a record to the worker list and keep tracking its status with a predefined interval. Transcoders will send the status of themselves and tasks running on them to the cluster manager periodically. The transcoder status includes CPU load, VideoCore IV load, temperature, and free memory. MQTT's *last will message* mechanism is used to implement the online status tracking of the transcoders. The cluster manager also embeds a web server to provide a dashboard for administrators to monitor the cluster's status and change the configurations. We have three design goals for the cluster manager: scalability, reliability, and elasticity.

Scalability: The cluster manager only maintains critical information of the transcoders. And only status information is frequently exchanged between the cluster manager and transcoders. Media servers manage the source videos and transcoded videos. And we can replicate the media server if its workload is too high. The separation of data flow and control flow ensures the cluster manager's scalability.

Reliability: We ensure the reliability of the system by real-time status monitoring coupled with low latency scheduling to migrate the failed tasks to working transcoders. The media server updates the index file on-the-fly to consistently list transcoded video streams. A temporary failure will not affect the video players.

Elasticity: We define priorities for tasks and transcoders. An important characteristic of adaptive bitrate video streaming is that every video program has multiple versions of encoded videos. The more variants of a video available, the more flexible

the players can optimize the user experience. Our system leverages that characteristic to implement an elastic transcoding service. When the available transcoders have enough resources to run all the transcoding tasks, all the tasks will be assigned to transcoders. But if not, only the high priority tasks will be scheduled and executed. We can easily extend the transcoder cluster by adding more transcoders with this elastic design to support more TV channels.

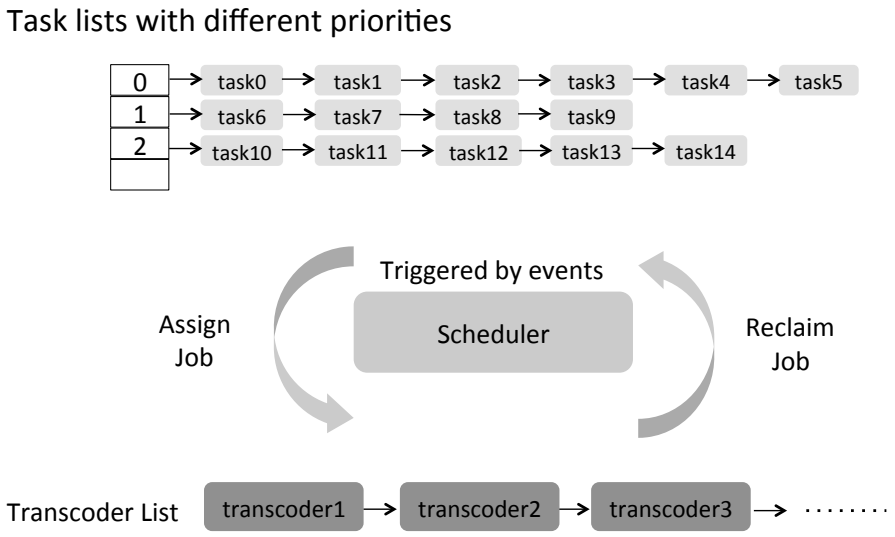


Figure 2.9: Overview of the cluster manager. The scheduler has an event-driven design.

Fig. 2.9 shows the internal data structures of the cluster manager. The scheduler makes decisions based on the capacities of the transcoders and the resource requirements of the transcoding tasks. The capacity and resource requirement are both positive integers. Their values determine which tasks and how many tasks can run on a transcoder in real-time.

Table 2.2 presents several task examples. A Raspberry Pi Model B's capacity is 20, so it can run one HD transcoding task (4 or 5), or two SD transcoding tasks (1 and 2), or three SD transcoding tasks (2, 3, and 7), or one HD transcoding task and 1 SD transcoding task (3 and 6). When a new transcoder registers to the cluster

ID	Channel	Command	Resource	Priority
1	A	transcode rtmp://192.168.1.172:1935/live/A_src rtmp://192.168.1.172:1935/live/A_800k 800	10	0
2	A	transcode rtmp://192.168.1.172:1935/live/A_src rtmp://192.168.1.172:1935/live/A_600k 600	8	1
3	A	transcode rtmp://192.168.1.172:1935/live/A_src rtmp://192.168.1.172:1935/live/A_400k 400	6	0
4	B	transcode rtmp://192.168.1.172:1935/live/B_src rtmp://192.168.1.172:1935/live/B_2400k 2400	20	0
5	B	transcode rtmp://192.168.1.172:1935/live/B_src rtmp://192.168.1.172:1935/live/B_1600k 1600	16	1
6	B	transcode rtmp://192.168.1.172:1935/live/B_src rtmp://192.168.1.172:1935/live/B_800k 800	14	0
7	C	transcode rtmp://192.168.1.172:1935/live/C_src rtmp://192.168.1.172:1935/live/C_400k 400	6	0

Table 2.2: Transcoding task examples

manager, the idle task in the highest priority task list will be assigned to it. When a task fails and returns to the cluster manager, the task manager will try to assign it to another transcoder. If the cluster manager can find a transcoder that has enough capacity for it, it will assign the failed task to that transcoder. If not, the cluster manager will try to revoke tasks with lower priorities from a transcoder and then assign this task to that transcoder. If the cluster manager cannot find a running task which has lower priority than the failed task, the cluster manager will not do anything. As discussed in Section 2.3, if we successfully reschedule a failed task, we need to send messages to transcoders to reset the RTMP connections of those streams corresponding to the same channel as the failed task.

Implementation

We implemented both the cluster manager and transcoders on the Linux operating system. We extensively used open source software in this project, including Apache [41], Nginx [42], node.js + Express [43], mqtt.js [44], paho MQTT client library [45], GStreamer, and Google protobuf. We used Mosquitto [46] as the MQTT message broker.

Media Server: We built the media server with Nginx + Nginx_RTMP_module [47] and Apache. The RTMP protocol is implemented by Nginx, whereas the HTTP interface is provided by Apache. The media server was installed on a server with Ubuntu 14.04 LTS.

Cluster Manager: The cluster manager is written in Javascript, and built on node.js + Express. We used mqtt.js to develop the MQTT client module that subscribes and publishes messages related to the transcoders. The cluster manager was also installed on a server with Ubuntu 14.04 LTS.

Transcoder: The transcoder's two components — transcoder worker and cluster agent are implemented in C/C++. They depend on GStreamer, paho MQTT client library, and Google protobuf. We built the SDK, root disk and Linux kernel for Raspberry Pi with buildroot [48], then we built the two components with the customized SDK.

Deployment

We deployed *VideoCoreCluster* in an incremental way. We leveraged a hybrid approach to provide transcoding service for the live video streaming service. The deployment had a small scale *VideoCoreCluster* with 8 Raspberry Pi Model Bs and a transcoder cluster composed of five powerful servers with Intel Xeon processors.

2.4 Evaluations

We evaluated *VideoCoreCluster* on video quality and transcoding speed. We also analyzed the power consumption of *VideoCoreCluster* and compared it with a transcoder cluster built with general-purpose processors.

Video Quality Test

The H.264 decoder module of VideoCore IV can support real-time decoding of H.264 high profile, level 4.0 video with resolutions up to 1920x1080 with very low

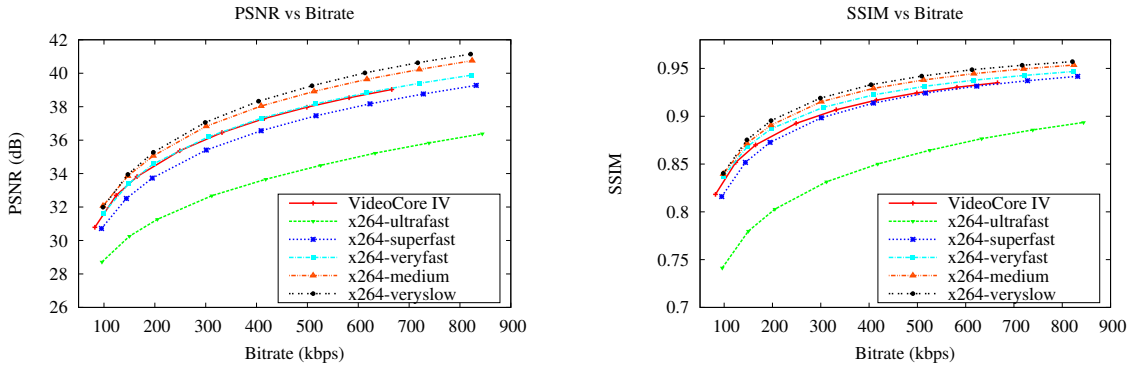
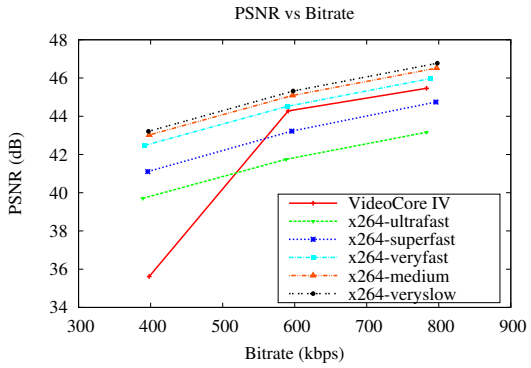


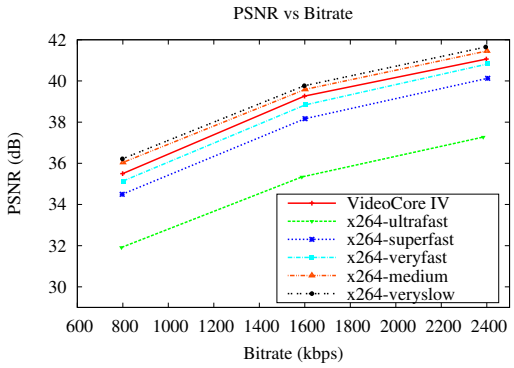
Figure 2.10: Video quality test results of VideoCore IV and x264 with different presets. We used *foreman*(352x288, 25fps) YUV sequence to test the encoders. We set IDR to 2 seconds, and disabled B-frame support of x264.

power consumption. In addition, the decoding result is exactly the same as the reference H.264 decoders. So the quality loss of our transcoding system is only from the encoder. Many hardware video encoders have some optimizations to simplify the hardware design while sacrificing video quality, especially for the video encoder modules in SoCs for mobile devices, because of the stringent restriction on power consumption. In order to have a clear idea about the video quality of the VideoCore's H.264 video encoder, we conducted tests on it and compared its performance with x264.

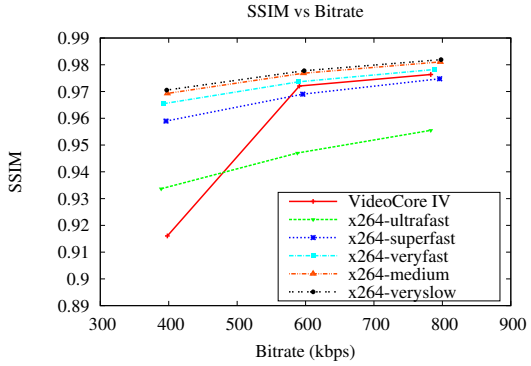
Both subjective and objective metrics are available for evaluating video quality. Subjective metrics are desired because they reflect the video quality from the users' perspective. However, their measurement procedures are complicated [49, 50]. In contrast, objective metrics are easy to measure; therefore, they are widely used in video encoder developments, even though there are arguments about them. Two metrics, Peak Signal to Noise Ratio (PSNR) and Structural Similarity (SSIM) Index, are widely used to compare video encoders regarding quality. PSNR is the traditional method, which attempts to measure the visibility of errors introduced by lossy video coding. Huynh-Thu et al. showed that as long as the video content and the codec type are not changed, PSNR is a valid quality measure [51]. SSIM is a complementary framework for quality assessment based on the degra-



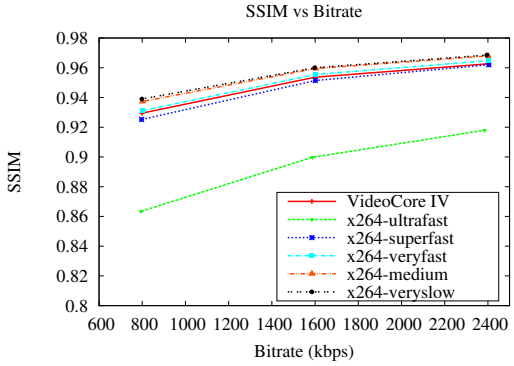
(a) PSNR values of an SD channel (720x480, 30fps)



(b) PSNR values of an HD channel (1280x720, 30fps)



(c) SSIM values of an SD channel (720x480, 30fps)



(d) SSIM values of an HD channel (1280x720, 30fps)

Figure 2.11: Video quality test results of VideoCore IV and x264 with different presets. We maintained the same configurations in the tests (IDR is 2 seconds, B-frame support of x264 is disabled).

dation of structural information [52]. We evaluated the hardware video encoder's performance with both the PSNR and SSIM.

The x264 has broad parameters to tune, which are correlated and it is hard to achieve the optimal configuration. Rather than trying different parameter settings, we used the presets provided by x264 developers to optimize the parameters related to video encoding speed and video quality. The x264 provides ten presets:

ultrafast, *superfast*, *veryfast*, *faster*, *fast*, *medium*, *slow*, *slower*, *veryslow*, and *placebo* (the default preset is *medium*). They are in descending order of speed and ascending order of video quality. As the video quality increases, encoding speed decreases exponentially.

We used two sets of video sequences to evaluate the encoders. The first one is the YUV sequences commonly used in video coding research [53], so the results can be easily reproduced and compared with other researcher's results. The second one was captured from our deployment that reflects the encoder's performance in the practical deployment. We natively compiled the x264-snapshot-20150917-2245 on a desktop with Ubuntu 14.04 LTS. We also cross-compiled the libraries, drivers and firmware for Raspberry Pi from [34, 35] on the same machine. One thing we noticed was that the H.264 encoder of VideoCore IV does not support B-frames. The reason is that the target applications of the SoC are real-time communication applications, e.g., video chatting and conference, where low latency is a strict requirement. B-frame leads to high encoding/decoding latency. Thus, the H.264 encoder of VideoCore IV does not support it. For a fair comparison, we tested x264 with and without B-frame support to check the impact of B-frame support on video quality vs. bitrate.

For all the video sequences we tested in the first set, we obtained similar results, though the exact numbers vary. We also found that the VideoCore's video encoder generated very low-quality video when the target bitrate was very low. That could be a bug of the video encoder's implementation. For brevity, we only show the results of *foreman_cif* here. We omit the results of x264 with B-frame because B-frame does not have a significant impact on the quality in our encoding settings. From Fig. 2.10, we can see VideoCore IV has similar or better performance regarding video quality compared to the x264 with *superfast* preset. Since the purpose of the second test set is to evaluate the video encoder's performance in practical deployment, we only evaluated the encoder's performance with the typical bitrates. Fig. 2.11a and Fig. 2.11c indicate that the VideoCore has poor performance on low bitrate settings. However, we believe it is not a big concern for deployment. When the players have to use that low bitrate version of the video streams, the player's

network performance must be very low. We do not expect that will be a common condition. As shown in Fig. 2.11b and Fig. 2.11d, VideoCore's video quality is good for high bitrate settings. Its quality is close to x264 with the *medium* preset.

Transcoding Speed Test

We measured the transcoder's performance under stress. We transcoded the video streams captured from an SD channel and an HD channel offline and recorded the time for transcoding. There are overheads on demuxing and muxing in the process, but because of the high complexity of transcoding (decoding + encoding), the bottleneck of the process is on the video transcoding. The detail specifications of the test videos are (1) SD channel: 720x480, 30fps, H.264 high profile, level 4.0, 1.2 Mbps. (2) HD channel: 1280x720, 30fps, H.264 high profile, level 4.0, 4Mbps. For some SD channels with lower resolutions in our deployment, the transcoding speed is higher than the results shown here. We transcoded the SD and HD videos to 800kbps and 2.4Mbps respectively and kept other parameters the same.

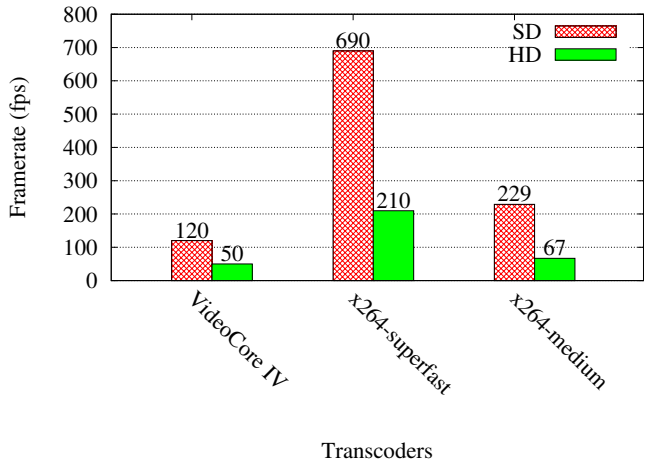


Figure 2.12: Transcoding speed of VideoCore IV and x264.

The hardware video encoder in VideoCore IV can support encoding 1920x1080, 30fps, H.264 high profile video in real-time. However, when we run the decoder

and encoder at the same time, the performance is not sufficient to support such high-resolution transcoding in real-time because the video decoder and encoder share some hardware resources. Even with the optimization described in Section 2.3, the transcoder can only support transcoding video with resolution up to 1280x720 in real-time.

We also measured software transcoder's speed for comparison. The software video transcoder we used is FFmpeg, which has built-in H.264 video decoder. We linked it with libx264 to provide H.264 video encoding. The desktop we used to run FFmpeg has an Intel Core i5-4570 CPU @ 3.20GHz and 16GB RAM. We built FFmpeg and libx264 with all the CPU capabilities (MMX2, SSE2Fast, SSSE3, SSE4.2, AVX, etc.) to accelerate the video transcoding. We tested *superfast* (similar quality as the video encoder of VideoCore IV) and *medium* (default) presets of x264. Fig. 2.12 shows that when the output video qualities are similar, software transcoder executing on the powerful Intel i5 CPU runs about 5.5x and 4x faster than the video transcoder running on Raspberry Pi for SD and HD channel respectively. For our transcoding system, which transcodes video in real-time, that means we can run 5.5x (SD) or 4x (HD) more transcoding tasks on a desktop than a Raspberry Pi. However, a Raspberry Pi is much cheaper and consumes much less power than a desktop with an Intel i5 CPU.

Power Consumption Analysis

We can see the superiority of Raspberry Pi regarding power efficiency from the transcoding speed test. An Intel Core i5-4570 processor has an average power consumption of 84W [54]. If we include the power consumption of other components, e.g., RAM and hard disk, the power consumption of a server would be higher than 100W. Raspberry Pi Model B in a configuration without any peripherals except Ethernet has typical power consumption about 2.1W [55]. The desktop consumes more than 40 times power than the Raspberry Pi Model B while it can do about 5.5x SD video transcoding or 4x HD video transcoding. We can see that the *VideoCoreCluster* is more energy efficient than a transcoder cluster built with

general-purpose processors to provide the same transcoding capacity. We omit the power consumption analysis on the network switches in our system deployment because we can use the same switches for different video transcoders.

2.5 Conclusion

High-quality video transcoding is critical to ensure high-quality video streaming service. We implemented *VideoCoreCluster*, a low-cost, highly efficient video transcoder system for live video streaming service. We built the system with commodity available, low-cost single board computers embedding high-performance and low-power video encoder hardware module. We implemented the cluster based on a network protocol for IoT for the control path and RTMP for the data path. This separation design achieved low latency in video transcoding and data delivery. Our system has much higher energy efficiency than a transcoding cluster built with general-purpose processors, and it does not sacrifice quality, reliability, or scalability. We can use *VideoCoreCluster* on other live video streaming services, and we can further improve the system on capability and energy efficiency by upgrading the individual transcoders.

3 Optimizing Real-time Video Communications at the Edge

3.1 Introduction

In the previous chapter, we introduced our work on a live video streaming service. The idea is using transcoders to generate multiple versions of compressed videos from one source video so that a video player can choose the best version to download and play. In this chapter, we introduce our work on another type of video application — Real-Time Communication (RTC). The system we built is too resource-intensive for mobile devices, and it requires low latency computation offloading. So, we have to deploy it on edge servers.

Modern smartphones are equipped with cameras and in-built applications for RTC, such as Skype, FaceTime, and Google Hangout. RTC is also used in remote monitoring, telemetry, etc. According to the Cisco Visual Networking Index, live Internet video will account for 17% of Internet video traffic by 2022 [56]. RTC video streaming poses a higher performance requirement than typical video playback applications. The real-time nature prohibits RTC from taking advantage of buffering, and many live video applications require extremely high responsiveness (hundreds of ms latency). These special requirements call for further optimization over the streaming process for enhanced user experience.

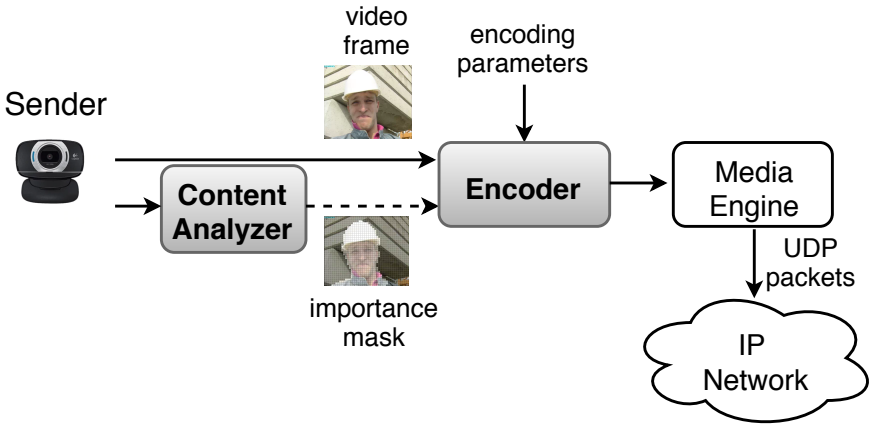
Unique characteristics of live video. Quite different from general videos streamed

over the Internet, RTC videos have two unique characteristics: *(i)* The viewer typically cares more about the foreground objects than the background objects. A typical foreground object would be a person's face in a live video chat, or a whiteboard in remote education/conference, and background can be the room environment that the person/whiteboard is in. *(ii)* RTC video frames are usually similar across time in typical scenarios such as video conferencing and video chat, where neither the foreground and background objects change significantly. In contrast, frames in a general video may go through drastic color and graphics changes frequently.

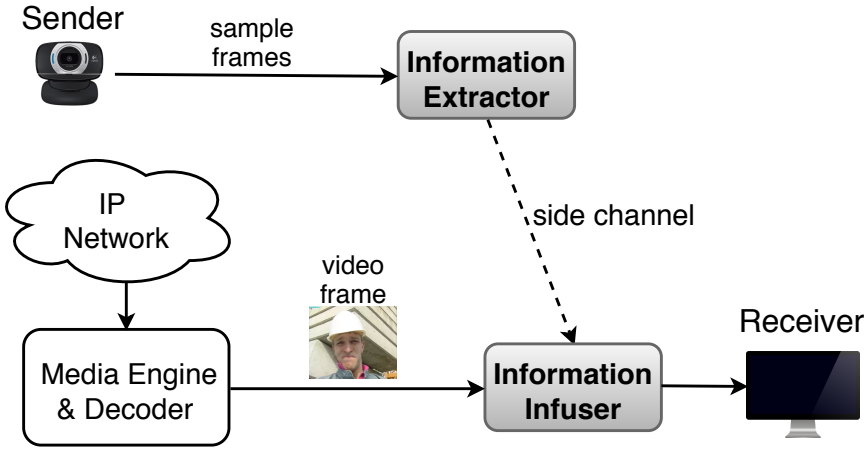
General-purpose video compression does not take advantage of high-level content information. Academic and industrial efforts have enabled standardized and commercially applicable video compression techniques, including MPEG2 [57], H.264 [58], HEVC [59], VP8 [60], VP9 [61], and the most recent AV1 [62]. While different in implementation details, these techniques take advantage of common characteristics in general videos and human perception, such as similarities between neighbor frames, and human visual systems are not sensitive to the high-frequency element of a signal. Designed to be general-purpose, these compression techniques are unable to take advantage of content information in specific applications. In addition, their design and implementation impedes possible optimizations in transmission mechanisms. For example, the information in I-frames cannot be shared by frames across instantaneous decoding refresh (IDR) frames, or frames in different sessions.

Exploiting deep neural networks to optimize video transmission. Deep neural networks (DNNs) have achieved remarkable performance on many computer vision tasks, such as image classification, object detection, and semantic segmentation. Our work exploits the power of DNNs to optimize video transmission with respect to RTC's two unique characteristics articulated above. We propose a hybrid approach — *enhance* traditional video compression techniques with lightweight DNNs. *DeepRTC* relies on two optimizations atop existing compression schemes:

1. *content-aware encoding:* at the sender, a DNN is used to analyze the video content to generate metadata, which is then utilized by the encoder to optimize



(a) Content-aware Video Encoding. Our prototype system uses object detection and semantic segmentation models to build the content analyzer.



(b) Video Quality Enhancement. We build our prototype system with super-resolution models but other models can be used as well, such as compression artifacts reduction models.

Figure 3.1: *DeepRTC* overview

bitrate allocation (Section 3.2).

2. *quality enhancement:* DNNs are trained with representative frames to extract information, which is then used at the receiver side to enhance the video quality (Section 3.3).

Content-aware encoding at the sender side. The goal of the content-aware encoding is to handle pixels with different importance with different priorities. It can reduce bandwidth usage without downgrading the quality of the important pixels, or improve the quality of important pixels without increasing bandwidth usage.

Quality enhancement at the receiver side. Quality enhancement approach divides the information flow into two parts: i) information common to all frames, and ii) information unique to every frame. Ideally, one would only transmit the unique information of every frame with the RTC channel. In the quality enhancement module, the first step involves extracting information from representative video frames. This is then shared with the receiver before the RTC session setup, which in turn enhances the received video frames. Fig. 3.1a and Fig. 3.1b show the two optimization approaches of *DeepRTC*. The gray boxes highlight the modules to implement the optimizations and DNNs are used to implement the system.

Distinctions between *DeepRTC* and previous approaches. It's worth noting that DNN-based end-to-end video compression have been recently proposed [63, 64]. Distinct to this line of research, our proposal does not use DNN to perform encoding directly. *Instead, DNNs guide the video coding and transmission.* Thus, *DeepRTC* is compatible with almost all encoding techniques including both DNN-based encoding and traditional approaches (MPEG2, H.264, HEVC, etc.).

Edge computing is required to build *DeepRTC* for mobile devices. DNNs are computationally expensive, therefore many mobile SoCs include the hardware accelerators to improve the performance and efficiency of DNNs executions, especially for the inference step [65]. However, based on Facebook's research [10], the performance gap of the CPU and the accelerators in the majority of mobile SoCs in the market is not significantly. Also, the programmability of those accelerators is a roadblock to employ them in mobile applications. In particular, for the DNNs used in the *DeepRTC* prototype, most of the SoCs cannot support the real-time inference. We believe the performance gap between mobile SoCs and hardware on servers will always be there because of the restrictions on power consumption and heat dissipation for mobile devices. So we have to build *DeepRTC* with the computation

offloading approach — offloading the DNN inference into a server with powerful hardware, e.g., GPUs, to support mobile devices. We choose to implement the offloading approach with edge computing, because edge computing can provide lower latency responsiveness and better privacy protection than cloud computing.

Another important reason that we have to use computation offloading to build *DeepRTC* is about the hardware video encoder. We choose VP9 as the codec for the prototype. Unfortunately, many mobile devices do not include hardware VP9 encoder and decoder, or do not provide the required interface for developers to tune the bit allocation. So we have to use a software VP9 implementation to build the *DeepRTC* prototype. The CPUs of mobile devices are not enough to guarantee real-time execution and we have to use a powerful edge server for that.

3.2 Content-aware Video Encoding

Traditionally, a video encoder treats all pixels in a frame equally in the encoding process. Some video encoders support encoding certain areas in a frame differently, namely with Region of Interest (ROI) encoding. Even though video encoders provide such a capability, it is not widely used in practice as there is no generally-applicable method to detect the region of interest.

The basic mechanism behind ROI encoding is a region-based adjustment of the quantization parameter (QP). This encoding optimization tunes parameters that are supported by unmodified decoders; it can be incrementally deployed in existing systems. Properly applied ROI encoding can reduce the number of bits required for video frames without downgrading the quality of important pixels or, alternatively, provide higher quality representations of the important pixels without increasing the number of bits.

This section provides background on quantization and ROI encoding followed by the design of our DNN-based ROI encoding, and intuitively shows the benefits of such an approach. Detailed evaluation of the system is given in Section 3.5.

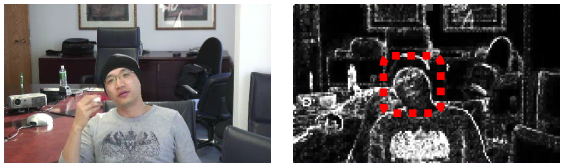
Quantization and ROI Encoding

Quantization is a major source of information loss in traditional lossy video encoders [66]. It changes the representations of the residual signal after transformation. Thus, quantization adjustment — adjusting the quantization parameters (QPs) is commonly used to implement ROI encoding. Larger QPs translate to larger quantization steps, which lead to more information loss. This implies that fewer bits are needed to represent the source data.

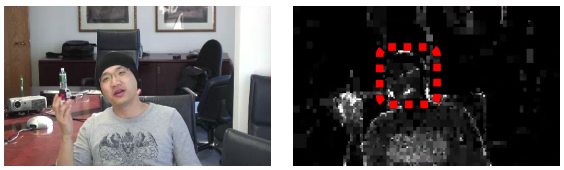
Different video encoders provide different granualarities to define QPs. For example, H.264 supports specifying different QPs for different slices [58]. By default, the VP9 encoder uses only one QP for the whole frame, but VP9 supports dividing a video frame into multiple pieces called segments; each segment can have a different QP and loop filter strength. QP values span from 0 to 63, where 0 corresponds to lossless quantization and 63 corresponds to the highest quantization step (and also the highest information loss). The general idea behind a video encoder's ROI support is providing an interface to set delta QPs — the amount of adjustment applied to the QPs generated by the rate control algorithms. Positive delta QPs result in lower accuracy of the quantization.

ROI encoding adjusts the QPs determined by the rate control module. Rate control is an important module in video encoders that constrains the rate of the encoded bitstream, while preserving the highest possible visual quality. Rate control provides various operational modes for different use cases, e.g., VBR (Variable Bit Rate), CBR (Constant Bit Rate), CQP (Constant Quantization Parameters) and CQ (Constrained Quality). For CBR mode, which is widely used in real-time applications, the adjustment of QPs for ROI encoding changes the feedback to the rate control module, and eventually, enters a stable loop. Therefore, we do not have to modify the rate control module for ROI encoding.

Even though ROI encoding leads to some overhead because of the additional ROI parameters, overall it reduces the number of bits used to encode a frame. Section 3.5 evaluates its impact on the compression ratio.



(a) A Keyframe



(b) A Non-keyframe

Figure 3.2: Bit allocation visualization of keyframe and non-keyframe. Lighter pixels use more bits and darker pixels use fewer bits. The dashed box represents the ROI in the video chatting application. The video is "vidyo4" from "Xiph.org Video Test Media" [67].

Bit Allocation Analysis

In order to obtain a sense of the gain achievable through ROI encoding, we analyzed the default bit allocation of the VP9 encoder. We encoded a source video with the VP9 encoder using the similar parameters as the default settings of the WebRTC reference implementation from Google [68]. We built a tool to scan the encoded bitstream, count the bits allocated to each of the 8x8 blocks, and then calculate the assigned bits for each pixel. Assuming the face is the ROI, Fig. 3.2a visualizes the result of a keyframe, and Fig. 3.2b visualizes the result of a non-keyframe. From these two figures, it is clear the VP9 encoder allocates a large portion of bits to represent pixels outside of ROIs. We observe that by tuning the bit allocation, an encoder can assign more bits to represent more important pixels.

ROI Detection for Content-aware Encoding

Given the capability of the encoder to adjust QPs for different areas in a video frame, we introduce the methods to detect ROI automatically in real-time. Researchers

have tried to detect ROI based on content. For example, Liu et al. [69] define human faces as ROIs and detect them using the direct frame difference and skin-tone information. Not only are these ROIs inaccurate, but they are also closely tied to the encoder's implementation. Salient object detection has attracted a lot of interest in the computer vision community [70, 71, 72], and it can be used to detect ROI for video compression. However, it lacks the flexibility we want.

Object detection is a well-researched problem in the computer vision community with over two decades of research. Recently, DNNs have been used to achieve state-of-the-art performance on this problem, with some methods such as YOLO [73] and SSD [74] supporting real-time inference. In our work, we choose DNN-based object detection models to build the ROI encoding solution because of the good balance between accuracy, flexibility and speed. We can create different ROI detectors, and easily integrate them with *DeepRTC*. Meanwhile, users can easily switch between different detectors. *DeepRTC* processes the ROI detection result and generates the ROI mapping in the granularity of macroblocks. For the VP9 encoder used in the *DeepRTC* prototype, ROI information defined for 8x8 macroblocks is expected.

We have implemented two example region detectors — face detector and human detector. These two detectors can be used in video chat and conferencing applications. Other detectors can be developed for application-specific requirements, e.g., a car detector, if users care about cars in the video stream.

Case 1: Face Detection. Generally speaking, human faces are obvious objects of interest in video conferencing and chatting applications. Face detection is a classic computer vision task and has been studied for decades because it is the first step of many computer vision tasks, e.g., face pose estimation and face recognition. Researchers have explored various techniques to implement face detection [75], including boosting-based [76], Haar cascades [77], and HOG (Histogram of Oriented Gradients) plus SVM approach [78]. More recently, researchers have begun to use DNN-based solutions to implement face detectors. These are significantly more accurate and robust than other face detectors [79]. Fig. 3.3a(1) gives an example of the results detected by an SSD-based face detector, and Fig. 3.3a(2) shows the macroblocks defining the ROI. It can detect the left face even though the face is

(1) (2)

(a) Face Detection

(1) (2)

(b) Human Detection

Figure 3.3: ROI detection examples

not toward the camera, whereas the Haar cascades (Viola-Jones) detector cannot detect it. To support quality optimization for regions including faces at arbitrary angles in *DeepRTC*, we choose the DNN-based solution to build the face detector. The DNN-based detector is computationally heavy, so GPU acceleration is required for real-time detection.

Case 2: Human Detection. We can also use DNN models, such as SSD [74] and YOLO [73], to detect humans in the video frame. They generate a bounding box around the people detected in a video frame. However, humans normally occupy a large area in a video frame. Therefore, a more fine-grained detection method is desired. Semantic image segmentation is a good option for our purpose because it can classify every pixel in a video frame to be inside or outside of the ROI (human). There are many research efforts on DNN-based semantic segmenation in recent years, e.g., FCN [80], SegNet [81] and DeepLab [82]. Based on a review on deep learning techniques applied to semantic segmentation [83], we chose DeepLab to

build the *DeepRTC* human detector. Fig. 3.3b(1) shows the result of a semantic segmenation, and Fig. 3.3b(2) shows the macroblocks defining the ROI.

3.3 Video Quality Enhancement

In video communication, the receiver gets video packets from the network, decodes them, and executes some post-processing (e.g. scaling) before displaying the video frames on a screen. We cannot modify the decoding step to improve video quality because the decoding process is strictly defined by standards, such as H.264 [58] and VP9 [61], and many mobile devices provide hardware support for them, which cannot be modified. Fortunately, the post-processing step is a good place for us to enhance the video quality. An obvious advantage of quality enhancement at the post-processing step is that we can use the optimization with any video codec. In this section, we introduce the motivation and architecture of the video quality enhancement based on the information extraction and infusion approach.

Motivation

An important observation of many RTC applications is that a large area of the video frame does not change much frame-by-frame. Imagine you are in a conference call with your colleagues remotely. A large area of a video frame will be the background which does not change much during a conference, or even in different conferences (assume the conference happens in the same conference room). Attendees may not change in the whole conference, but they may change positions and poses. In video coding terminology, the duplicated information in different frames is temporal redundancy and inter-prediction is the mechanism to remove it [66]. The video encoder generates I-frames (only intra-prediction) and P-frames (inter-prediction is allowed) depending on the prediction mode. A P-frame is much smaller than an I-frame in practice because of the temporal redundancy reduction. For RTC, we always want to avoid I-frames if possible to reduce the bandwidth usage, but senders must transmit I-frames in case of packet loss, a new user joining, etc. In

addition, the information of an I-frame cannot be reused by receivers in a session, not to mention across sessions.

This observation motivates us to explore the idea to split information of a video into two parts: (i) information common among all frames, and (ii) information unique to every frame. If a sender transmits the shared information to receivers before the RTC session or during the session with some side channel, then all the receivers can infuse the information to all the received frames to enhance the quality. Ideally, only the unique information of every frame needs to be transmitted during the RTC session, and bandwidth usage is reduced.

Enhance Video with Super-Resolution

Many devices are equipped with screens larger than video resolutions used in RTC. Video resolutions up to 1280x720 are commonly used, but the current mainstream phones in 2018 have screen sizes up to 1920x1080. So upscaling is needed when a user wants to view the video in fullscreen. Unlike a simple scaler, such as bicubic [84], super-resolution (SR) approach can infuse information into the upscaled image to enhance visual quality.

Super-resolution, especially Single Image Super-Resolution (SISR) is a classic computer vision problem. Since the introduction of SRCNN [85], there are many research efforts on it with DNNs because of the superior performance. For example, FSRCNN [86], ESPCN [87], VDSR [88], DRCN [89], DRRN [90], LapSRN [91], SRDenseNet [92], SRGAN [93], EDSR [94], CARN [95], and many others. Some researchers also proposed to use DNNs to implement SR for videos [96, 97].

A baseline SR model trained with general images only has general knowledge (regarding images in the training dataset). If we use it directly to upscale video frames, the quality enhancement will be limited. However, on the sender side, we can fine-tune the baseline model with *representative* frames of a video sequence (with transfer learning) to inject information of the specific video sequence into the model, or, explained differently, the model can extract information from the video sequence and store the information in the model parameters. Then on the receiver

side, we apply the fine-tuned SR model on the decoded video to infuse information and enhance the video quality. The key to the quality improvement is that we need to make sure the video frames used in transfer learning is representative, which is not hard to guarantee because of the characteristics of the video data in many RTC applications. We also found the models trained for specific qualities (quality-aware) can further boost video quality. The fine tuning with transfer learning can be done at the sender, an edge server, or the cloud.

Challenges Using SR on RTC

Even though many SR models are available, only several models can be used in RTC because of the real-time inference and small model size requirements. We want a small model size because the model needs to be transmitted to the receiver, but the model size cannot be too small, because it must have enough capacity to store general knowledge of images as well as the extracted information from the representative video frames. For example, FSRCNN [86] is a very small SR model with acceptable performance, but it is too small to carry video information. Based on the requirements, we choose CARN [95] to build *DeepRTC* prototype. CARN provides a good tradeoff between inference speed, model size, and video quality improvement.

Yeo et. al propose using content-aware DNNs to build adaptive video streaming services [98]. They use overfitted DNN models (content-aware) to upscale video to get quality improvement. In their case, the video frames as the SR model input are in good quality (only in low resolution), so the content-aware SR approach is enough. For RTC, in many cases, the video quality is poor and has visible compression artifacts. As our evaluation indicates, when the information loss introduced by the lossy video encoder is too much, the information infusing through SR can only achieve small gains. To overcome the decay of the return in SR for low-quality frames, we introduce a quality-aware SR model to upscale the video.

SR Models Training and Transmission

The baseline SR model is trained with general datasets first. That training can be done offline only once with a large dataset, and the model can be distributed to receivers before any RTC sessions begin. The content-aware and quality-aware SR models must be trained somewhere (in the sender itself, edge servers, or in the cloud) with representative video provided by senders to extract information of the video sequence. The representative video is very short (several seconds), and transfer learning can be done relatively quickly (several minutes).

Since the content-aware and quality-aware SR models are very small (several MBs in the prototype), they can be transmitted before a communication session starts and cached in the receiver side for future use.

Real-time Inference of SR Models

Unlike many other DNNs (for image classification, object detection, etc.) which shrink the input data layer-by-layer and the output is much smaller than the input, SR models inflate the data size, and the intermediate layers also require large buffers, which makes the SR model inference a computationally intensive and memory-rich operation. So a hardware accelerator such as a GPU is required to guarantee real-time inference, and the hardware must provide enough memory to run the SR model.

3.4 Implementation

We build the *DeepRTC* prototype on a desktop computer with 3.1GHz Intel i5-3350P CPU, 16GB system RAM, an NVidia 1080 GPU with 8GB RAM. We installed Ubuntu 16.04 LTS and CUDA 9.0 on the desktop. For *DeepRTC* deployment on mobile devices with edge computing approach, we assume the mobile device has high bandwidth wireless network connection to the edge server, and the latency is very low (<10ms). That is true for most home networks and will be practical in the

cellular network after the 5G network is deployed. As a result, we can compress the video with minimal quality loss (or even lossless) because of the high bandwidth edge network. So we ignore the details of the deployment with the mobile client and edge server, and focus on the *DeepRTC* prototype on a desktop computer.

We build *DeepRTC* with libvpx 20180706 snapshot [99], OpenCV 3.4.1, PyTorch 1.0, and TensorFlow 1.9.0. We modified the source code of the VP9 encoder in libvpx to enable ROI encoding for keyframes because the gain is very low if the ROI encoding is only applied to non-keyframes. We choose a MobileNet SSD (single shot multibox detector) based object detection model [100] to implement the face detector, and the DeepLab v3 TensorFlow implementation [101] to build the human detector. We build a quality enhancement approach based on the CARN SR model [95]. We want to build a system that can be easily integrated into real-time video applications. So we implement the system with C++ and the integration with PyTorch and TensorFlow is through the corresponding C++ APIs. We also build an end-to-end RTC application with the GStreamer framework and the WebRTC plugin [102]. We do not choose libwebrtc [68] because of the flexibility of the GStreamer framework to integrate video codecs and optimization modules into the video pipeline.

3.5 Evaluations

In this section, we report results of our evaluation of the two optimizations of *DeepRTC*, namely content-aware encoding and quality enhancement. We ran all our test on the hardware introduced in Section 3.4. We evaluate the optimizations separately to examine individual gains, as opposed to benchmarking end-to-end system performance.

Test Setup

A WebRTC implementation in practice, e.g., the reference implementation from webrtc.org, has a sophisticated strategy to decide configurations of the video encoder,

Number	Video Name	Resolution	Frames
1	deadline	352x288 (CIF)	1374
2	students	352x288 (CIF)	1007
3	vtc1nw	720x480 (NTSC)	360
4	vidyo1	1280x720 (720p)	302
5	vidyo4	1280x720 (720p)	302
6	Johnny	1280x720 (720p)	302

Table 3.1: Collection of videos used in our evaluations.

such as speed, resolution, framerate, bitrate, and frame dropping policy. To ease the evaluation, we only test the system with simplified and typical configurations.

Since the typical applications of *DeepRTC* are video chatting and conferencing, we use video sequences containing humans for the testing of our system. We chose six videos from "Xiph.org Video Test Media" [67], including both low-resolution and high-resolution to encapsulate greater diversity. All these videos were recorded with static cameras, and we include videos with both a single person and multiple people. Table 3.1 gives detailed information of those test videos. We converted the frame rate to 30FPS (frames per second) for the videos with 60FPS frame rate to ensure that all test videos have the same frame rate. We also scale vtc1nw from 720x486 to 720x480 because the content-aware encoding of the *DeepRTC* prototype only supports video resolutions which are multiples of 8. All videos are converted to YUV420 format before the test.

We use both the peak-signal-to-noise ratio (PSNR) and the structural similarity index (SSIM) [52] as the quality metric. Additionally, we built a tool to calculate the PSNR and SSIM values of ROI blocks and non-ROI blocks. By default SSIM is in the range [0, 1.0]. It is hard to compare SSIMs directly, so we convert it to dB with the formulation below. Note that the same formulation is used in x264 and libvpx internally.

$$\mathrm{SSIM}' = 1 - \mathrm{SSIM}$$

$$\mathrm{dB} = \begin{cases} -10 \times \log(\mathrm{SSIM}') & \mathrm{SSIM}' > 10^{-10} \\ 100 & \mathrm{SSIM}' \leqslant 10^{-10} \end{cases}$$

The key takeaways from our evaluation include:

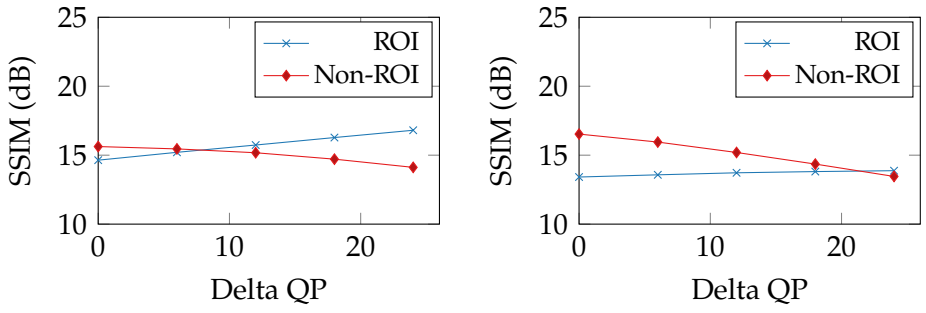
1. Depending on rate control mode, content-aware encoding can either improve the quality of important pixels up to 2dB in terms of SSIM, or reduce bandwidth usage up to 50%.
2. The video enhancement approach can improve video quality up to 2dB in terms of SSIM, 5dB in terms of PSNR.
3. Both optimization approaches can execute in real-time.
4. The model sizes are small enough (tens of megabytes) to be efficiently deployed, and the GPU memory required to run the inference is within those provided by mid-tier GPUs.

Content-Aware Encoding

Video Quality Improvement. When *DeepRTC* operates in CBR mode, it can improve the quality of ROI blocks by sacrificing the quality of non-ROI blocks. *DeepRTC* can only achieve gains when the QPs decided by the rate control module are not too small or too large. We chose 1800kbps as the target bitrate and the QPs of the encoded frames are in the middle of the range.

Fig. 3.4a and Fig. 3.4b compare the SSIMs of ROI and non-ROI in the compressed video (vidyo4, 1800kbps) when the face detector and the human detector are used respectively. These figures indicate that a higher delta QP leads to greater quality improvement on ROI blocks. They also show the video quality when the ROI encoding is not enabled (Delta QP = 0). In our test, *DeepRTC* can improve the video quality of ROI blocks as much as 2dB (in terms of SSIM and PSNR) when the delta QP is 24.

Bitrate Reduction. When the video encoder works in the CQ mode, *DeepRTC* can reduce the overall video bitrate by reducing the bits used to represent non-ROI blocks. Fig. 3.5 show the results of encoding "vidyo4" with different ROI detectors. The target bitrate is 1800kbps and the baseline QP is 27. *DeepRTC* can reduce bitrate



(a) face detector (b) human detector

Figure 3.4: SSIM of vidyo4 with different delta QPs

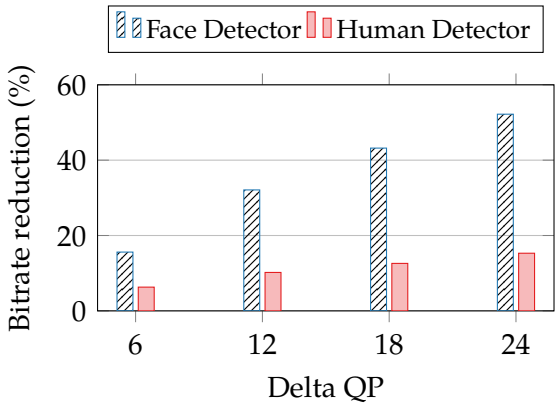


Figure 3.5: Percentage of bitrate reduction for vidyo4 with different delta QPs.

by as much as 50% without quality loss in ROI blocks. We also test the bitrate reduction when the keyframe interval varies. and find that the keyframe interval does not impact the bitrate reduction much.

Video Quality Enhancement

The quality enhancement module of *DeepRTC* infuses information into the video frames with DNN-based super-resolution to improve video quality. This section presents the results on three test videos.

Because of the complexity of the SR model, the hardware platform of *DeepRTC*

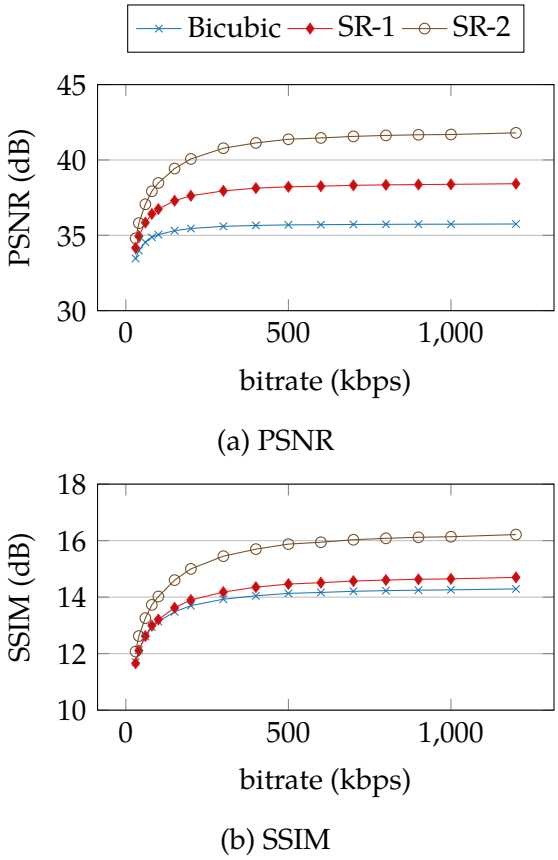


Figure 3.6: Quality of upscaled "Johnny" videos with different upscaling approaches (SR-1: baseline SR model. SR-2: content-aware SR model).

prototype only supports direct upscaling video frames from 320x180 to 640x360 in real-time. We downsample vidyo1, vidyo4, and Johnny to resolutions 640x360 and 320x180 and use them as the test videos. We reserve 90 frames (3 seconds) at the beginning of these videos to train the content-aware SR models, and compare the video quality improvement of different approaches with the remaining frames. The video pipeline consists of the video encoder, decoder, and scaler. The resolution of the input video is 320x180, and output video of the pipeline is 640x360. We have the original 640x360 video as the reference to calculate the video quality metrics (PSNR and SSIM).

In the first test, we evaluate the quality improvement of the baseline SR model

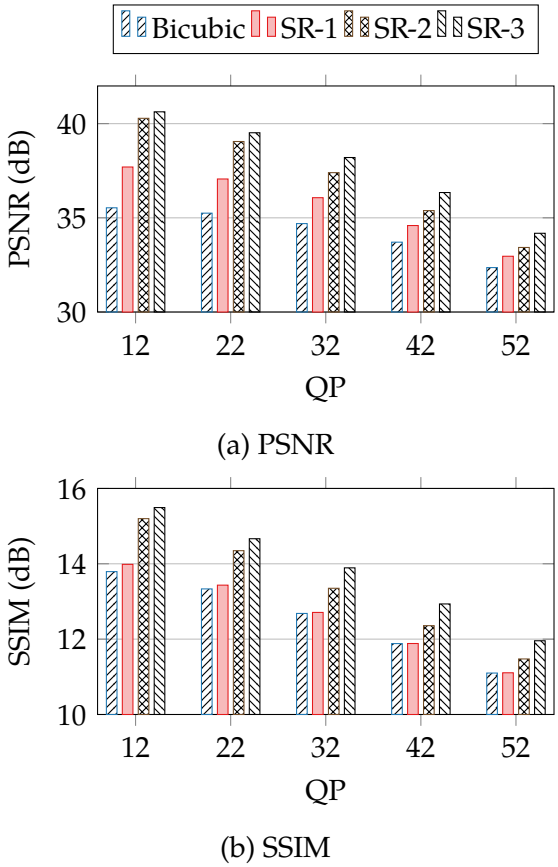
(trained with DIV2K dataset [103]) and the content-aware model (trained with the 90 frames of the test videos respectively with transfer learning) over the simple upsampling algorithm. The encoder works in CBR mode and compresses the videos to different bitrates during the test. The results are similar in all the test videos, so we only report the results of one test video in the figure. Fig. 3.6a and Fig. 3.6b show that both the baseline SR models and content-aware models improve the video quality by infusing information into the video frames. The baseline SR model can improve video quality up to 2dB in terms of PSNR, but the improvement in terms of SSIM is not much. The content-aware SR model can improve video quality in terms of both PSNR and SSIM. An interesting finding is that the improvement is low when the bitrate is low. Our hypothesis is that when the bitrate is low, the compressed video has already lost a substantial amount of information, and the SR models do not have enough hints to infuse information into the frames.

We probe this hypothesis in another set of tests. We encode the video with different QPs, and we do observe that lower the QP (higher quality), higher the gain we can get with the SR models. Based on this finding, we train SR models for different quality levels for every video sequence separately. The QP values have a range [2, 52]. We divide the range into five intervals, and use the values 12, 22, 32, 42, 52 to represent the intervals. In the test, the quality-aware SR model can further improve the video quality beyond the content-aware SR model. Fig. 3.7a and Fig. 3.7b show the results.

Despite the benefit in video quality, it may be difficult to use the quality-aware models in practice without further optimization. In order to guarantee real-time inference, it is necessary to load all five models into memory for the duration of the video session. That is difficult to do with current hardware because of the high memory requirement.

Speed Test

ROI Detection Speed. Both face detector and human detector are built with TensorFlow and they use GPU to accelerate the inference. For face detection, we choose an



(a) PSNR

(b) SSIM

Figure 3.7: Quality of upscaled "Johnny" videos with different upscaling approaches (SR-1: baseline SR model. SR-2: content-aware SR model. SR-3: content and quality-aware SR model).

SSD-based model built with TensorFlow [100]. For human detection, we choose the simplest pre-trained DeepLab model in order to support real-time detection [101]. We export and freeze the model from the checkpoint *mobilenetv2_coco_voc_trainaug* and set output stride to be 16. The inference for a frame should be done within 33ms in order to support real-time detection on source video with 30FPS frame rate.

In the first execution of a deep learning model, TensorFlow needs to optimize the neural network graph and initialize the GPU context to execute the neural network. As a result, the time taken to detect the region of interest in the first video

Figure 3.8: ROI detection time on frames excluding the first ones (median, minimum, and maximum).

frame is significantly longer than subsequent frames (between 1 and 3 seconds in tests). Fortunately, this time is negligible in practice because we can "warm up" the detector before the application starts streaming video.

Fig. 3.8 shows the median detection time for all the frames except the first one. We can see that the median detection time is far below 33ms, and the longest time is shorter than 33ms. Also, the ROI detection time is not very sensitive to video resolution.

The selected deep learning models provide the required performance metrics on our hardware platform for real-time region detection. If we choose other more complex deep learning models for ROI detection, we may have to use optimized inference engines, such as TensorRT [104] from NVidia, to accelerate the inference. We can also skip some video frames for detection when the video content does not change much frame-by-frame.

Encoding Speed. We build the VP9 encoder from source code with all processor specific optimizations enabled. The encoding parameters are similar to the ones in the reference WebRTC implementation [68]. We set the "cpu-used" parameter to 5 and the encoder works in "real-time" mode. The target bitrates of these compressed

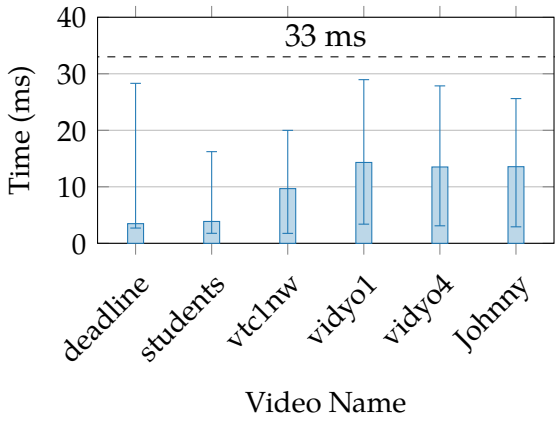


Figure 3.9: Encoding speed (median, minimum, and maximum).

videos are 400kbps, 1200kbps, and 1800kbps for resolution 352x288, 720x480 and 1280x720, respectively. We run the encoder with test videos and record the time taken to encode every frame. We can see the encoding time for frames in Fig. 3.9, has a very high variation for the frames in the same video. For some frames in the videos with resolution 1280x720, the sum of ROI detection time and encoding time is higher than 33ms. That means we cannot run those two steps sequentially to support real-time execution. Instead, ROI detection and the encoding processes must execute in a pipeline style.

Super Resolution Speed. The SR model is more complex in execution than the ROI detection models. Our hardware platform can only support real-time processing of videos with 320x180 resolution. Fig. 3.10 shows the result. The speed is just enough for real-time execution.

DNN Training Speed. The *DeepRTC* prototype uses multiple DNNs, and training of these DNNs is very time-consuming. For example, the baseline CARN model training with DIV2K dataset takes 75 hours for 600000 iterations. Fortunately, those DNN models can be trained offline, and *DeepRTC* only requires fast inference except for the information extraction step of the video quality enhancement approach. The information extraction step uses transfer learning. In our test, 1000 iterations can extract enough information from the video frames, which takes about 4 minutes.

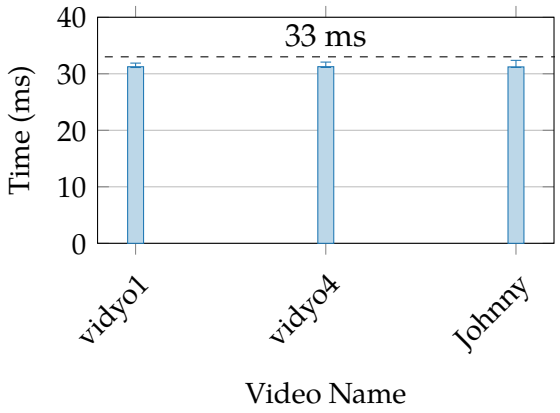


Figure 3.10: SR speed (median, minimum, and maximum).

Model	Model Size (MB)	GPU Memory Usage (MB)
Face Detection	22	7648
Human Detection	8.4	7656
Super-Resolution	3.8	1695

Table 3.2: Storage resource usage of the DNN models

Furthermore, the information extraction step does not need to run frequently, because the trained model can be reused if the video content does not change much. For example, if a user has a video conference in the same office many times, then she only needs to train the model once for all the conferences.

Resource Requirement of DNN Models

Table 3.2 shows the memory usage of these DNNs. The face detection and human detection models are TensorFlow model in Protocol Buffers format, while the super-resolution model is a serialized TorchScript. The sizes of these models should not pose an issue for deployment, though we need to note that the super-resolution model may need to be transmitted between users. The GPU memory usage in the table is the maximum value observed during inference. The values are not the minimum requirement, and they can be different if we run the inference with different GPUs.

3.6 Discussion

Our prototype system only supports VP9 codec, but the idea is general and can be used with other codecs, such as VP8 and H.264.

The Compression Artifact Reduction (CAR) [105] technique can also be used to infuse information into the video frames at the receiver when the upscaling is not necessary at the receiver. The system architecture will be the same as the *DeepRTC* prototype. We plan to explore a better quality enhancement approach by building a DNN which can accept reference video frames as input during inference, and eliminate the transfer learning step completely.

Mobile device vendors continue to improve the camera systems in both software and hardware, and some of the new features can be used to optimize video encoding. Apple's recent iPhones are equipped with dual cameras and support depth-of-field effects. We envision such kinds of new capabilities being used to optimize video encoding for real-time applications.

3.7 Conclusion

In this chapter, we introduce *DeepRTC* for RTC applications. Our proposal relies on a content-based video encoding optimization approach and a video quality enhancement scheme. The encoding optimization uses ROI encoding to put *more important* information in the packets, while the quality enhancement scheme adopts the idea of extracting information from video data at senders and using the extracted information to enhance video quality at receivers. We evaluate *DeepRTC* with publicly available video sequences. Depending on rate control models, the encoding optimization can improve the video quality of ROI blocks as much as 2dB in terms of SSIM and PSNR without increasing the overall bitrate, or reduce bitrate as much as 50% without quality loss on ROI blocks. The improvement of the quality enhancement approach is content dependent, and we found the highest video quality improvement is almost 2dB in terms of SSIM and 5dB in terms of PSNR. In

our research, we focus on the optimization of encoding and decoding. We plan to work on congestion control optimization for RTC in future work.

4 Computer Vision Algorithm Offloading to the Edge

4.1 Introduction

We introduced two applications leveraging the power of edge computing to optimize video transmissions in the previous two chapters. In this chapter, we explain our work on using edge computing paradigm for intelligent video analytics on mobile devices. The goal of this work is not for a single application, but for a group of applications requiring low latency computation offloading. For example, we can use the work described in this chapter to implement the *DeepRTC* system introduced in Chapter 3.

Deep learning with Deep Neural Networks (DNNs) is a hot topic in both academia and industry. It is widely used in different areas, such as speech recognition, language translation, image recognition, and recommendation. DNN-based approaches have achieved big improvement in accuracy over traditional machine learning algorithms on computer vision tasks. They even beat humans at the image recognition task [106]. In this chapter, we present our work on building a real-time intelligent video analytics framework based on deep learning for services deployed at the Internet edge.

Nowadays, camera sensors are cheap and can be included in many devices. But the high computation and storage requirements of DNNs hinder their usefulness

for local video processing applications in these low-cost devices. For example, the GoogLeNet model for image classification is larger than 20MB and requires about 1.5 billion multiply-add operations per inference per image [107]. At 500MB, the VGG16 model is even larger [108]. It is infeasible to deploy current DNNs into many devices with low-cost, low-power processors.

A commonly used approach to using deep learning in low-cost devices is offloading the tasks to the cloud. Amazon Alexa, for example, uploads processed audio data to the cloud to implement speech recognition and Natural Language Processing (NLP). Deep learning has two steps: training and inference. Cloud computing fits the requirements of deep learning training very well because the training process requires a large amount of data and high throughput computing. However, the inference process, which is used in intelligent video analytics, requires low latency that cloud computing, in many cases, cannot provide. Also, continuously uploading high-bandwidth video data to cloud servers wastes network resources. If many people are uploading video data simultaneously, Internet congestion could occur. Moreover, if the network connection is interrupted, the services relying on cloud computing will be down. Finally, the data used in inference is generally more sensitive than the training data. Based on these observations, we believe cloud computing is not appropriate for live video analysis. It is ideal to analyze the live video stream locally if possible.

Edge computing is proposed to solve those problems in cloud computing. By deploying shared resources at the Internet edge and pushing computation close to the data sources, edge computing can benefit many applications requiring low latency and high privacy. We propose an edge service framework for real-time intelligent video analytics — *EdgeEye*. We name the framework *EdgeEye* because it is like the eyes of third-party edge services and devices. It helps them to ***see*** and ***understand*** the ambient environment through the video signal. When compared with cloud computing solutions, *EdgeEye* uses significantly less bandwidth because the system does not need to upload high bandwidth video to the cloud continuously. It also further reduces the total computation resource consumption because there is no need to encrypt the video when the analysis happens locally.

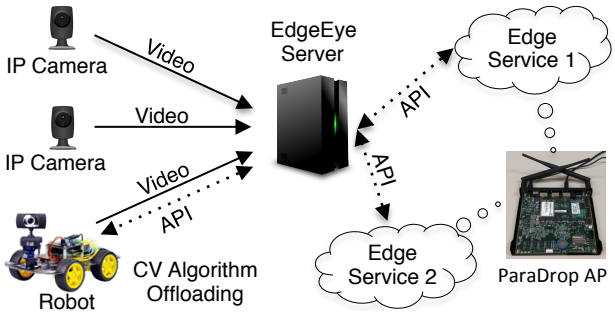


Figure 4.1: Overview of the *EdgeEye* deployment in a home environment. All the devices are on the same home network and can connect to each other directly. Two edge services and the robot can offload real-time video analysis tasks to the *EdgeEye* server using the *EdgeEye* API and get results with low latency.

ParaDrop [8] is used in our work to manage the edge services using the *EdgeEye* API. *ParaDrop* is an edge computing platform built with WiFi access points. It provides an API to manage the edge nodes as well as edge services running in the edge nodes. We can also use other edge computing platforms like resin.io [109] for the same purpose. The *EdgeEye* API provides a high level abstraction to hide the details of diverse machine learning frameworks like Caffe [110], TensorFlow [111], Torch [112], and MXNet [113]. The API covers different use cases, e.g., object detection, face verification, and activity detection. We implement the API with WebSocket protocol for low latency. *EdgeEye* uses a modular and extensible design, and we build the system with an open-source media framework — GStreamer [39]. We can easily reuse many open-source plugins of GStreamer to build complex pipelines.

Current research efforts on deep learning for computer vision focus more on training than inference. The inference performance will not be optimal if we directly use the deep learning frameworks to execute a DNN. Chip vendors like Intel [114] and Nvidia [104] provide optimized inference engines for CPUs and GPUs. *EdgeEye* leverages those implementations to implement highly efficient inference services.

Fig. 4.1 gives a high-level overview of the *EdgeEye* in a home environment. Two example edge services are deployed on the *ParaDrop* access point (AP). The *EdgeEye*

server has powerful CPU and GPU to analyze live video streams in real-time. The IP cameras in this figure can be standalone wireless cameras or cameras of other devices, e.g., baby monitors. Fig. 4.1 also shows a robot with a camera. With *EdgeEye*'s capability of real-time video analytics, the robot can see and understand the environment and then make decisions accordingly. The cost to build the robot will be reduced because it does not need to have a high-performance processor. The edge services and the robot use the *EdgeEye* API to manage video analysis sessions with given video sources. They specify parameters and deep learning models to be used by the analysis session. And the analysis results will be sent back to them in real-time.

4.2 System Design

Usually, developers build deep learning applications with various open source deep learning frameworks, e.g., Caffe, TensorFlow, and MXNet. These frameworks provide a wide variety of APIs and language bindings. We believe that providing uniform task-specific interfaces will simplify developers' work. Computer vision related tasks always include some supportive tasks, such as image input/output, decoding, image rescaling, and format conversion. Application frameworks need to provide such primitives with optimized performance, so that developers need not integrate and optimize other third-party libraries for these supportive tasks.

EdgeEye Architecture

Fig. 4.2 gives a high level overview of the *EdgeEye* framework. *EdgeEye* needs to be installed on a machine with high-performance CPU and GPU to guarantee real-time processing. It adopts the GStreamer framework to manage the video analysis pipeline and reuses open source elements to handle some supportive tasks, such as format conversion and rescaling. The *EdgeEye* elements are implemented based on the inference engines from chip vendors for the best possible performance and efficiency. Along with the GStreamer pipeline architecture, *EdgeEye* provides a

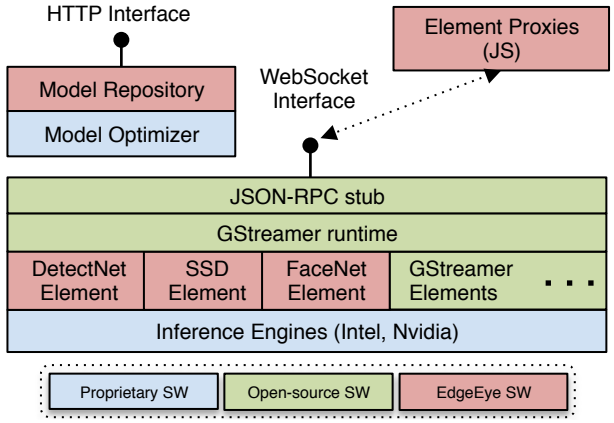


Figure 4.2: An overview of the *EdgeEye* software architecture. We extensively use open source components to build *EdgeEye*. The different colors indicate the different sources of the modules.

WebSocket API based on JSON-RPC and also element proxies to control *EdgeEye* elements through the API. *EdgeEye* also provides an HTTP API to manage deep learning models used by *EdgeEye* elements. The model repository integrates the model optimizers, which profile the deep learning models from deep learning frameworks, such as Caffe and TensorFlow, and generate the deployable files that the inference engines can load and execute directly.

DNN Model Management

Deep learning models store the structure, weights, and biases of DNNs. These models are often tens or even hundreds of megabytes in size. Different deep learning frameworks store them in different formats. For example, a deep learning model built with Caffe has a *.prototxt file and a *.caffemodel file, while the same model built with TensorFlow has one *.pb file. The model management module manages the models used by the *EdgeEye* GStreamer elements. When applications upload a new model to the model repository, the optimizer will profile the model and generate the optimized deployable model file for the corresponding *EdgeEye* element.

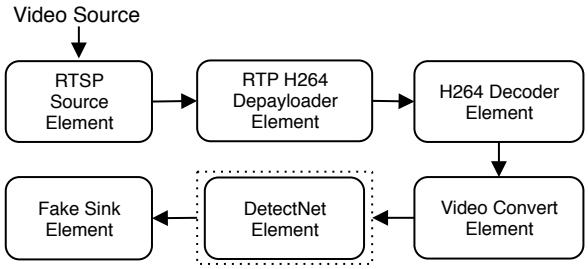


Figure 4.3: An object detection pipeline including the DetectNet element.

Video Analysis Pipeline

We use the GStreamer media framework to implement the video analysis pipeline. A pipeline is a chain of elements. Every element besides the source element processes video data received from the element before it. It then either forwards the processed data to the next element in the pipeline or discards the data. The pipeline and plugin mechanisms available in GStreamer simplify the creation of complex media processing pipelines. Fig. 4.3 shows an example pipeline to detect objects with DetectNet neural network. The non-DetectNet elements are from the GStreamer community and handle data reading, parsing, and decoding. The DetectNet element (inside the dashed box) executes deep learning models in the Nvidia GPU inference engine using the TensorRT library [104].

EdgeEye Element Design

EdgeEye elements are model and inference engine specific, i.e., we have SSD-INTEL, SSD-NVIDIA, YOLO-INTEL, and YOLO-NVIDIA elements for SSD (Single Shot multi-box Detector) [74] and YOLO [73] DNNs running on the Intel and Nvidia inference engines. *EdgeEye* hides the difference of these DNN models in network structure and input/output format. Based on the message mechanism of GStreamer, *EdgeEye* exposes the functionality through a task-specific interface, so that developers can get the video analysis functionality with minimal effort. Developers need to specify the model files (network structure, weights, and biases) to load into the inference engine, as well as specify other model-specific input and output

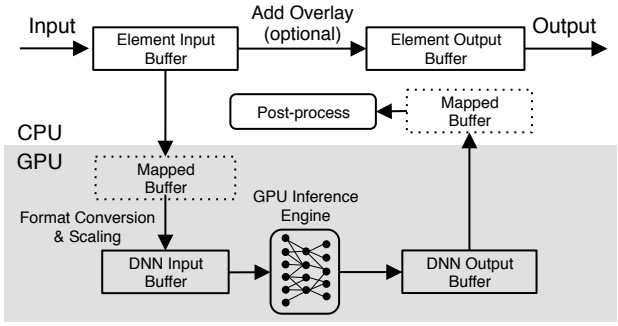


Figure 4.4: Internal of the DetectNet element.

parameters. The output of the DNN model can either be embedded in the video stream (overlay), or sent to pipeline manager through messages. These messages can then be forwarded to applications through the JSON-RPC interface.

DetectNet is an example DNN architecture in the Nvidia Deep Learning GPU Training System (DIGITS) [115]. Its data representation is inspired by YOLO. DetectNet uses GoogLeNet internally with some modifications. Similar to YOLO, it overlays an image with a regular grid. The grid has spacing slightly smaller than the smallest object we wish to detect, and the DNN generates information for every grid square. The method to generate output bounding boxes and class labels is totally different from YOLO. As shown in Fig. 4.4, we have to execute the post-processing steps including bounding box clustering to generate the final result. More detailed information is available in [116]. Fig. 4.4 also shows the buffer management in the element implementation. We use Nvidia CUDA mapped memory feature to simplify the memory movement between CPU and GPU. The DetectNet element uses TensorRT to control the GPU Inference engine, such as initialize the inference engine, create inference runtime, load the serialized model, create inference execution context, and execute the inference engine.

The hardest parts of an element implementation are in the buffer management and post-processing, both of which are on the CPU side. If TensorRT supports all the layers of a DNN model, we can easily load the serialized model and create the execution context. Unfortunately, TensorRT does not support some layers, e.g.,

Leaky ReLU, which is used in YOLO. We use the TensorRT plugin API to implement those unsupported layers ourselves.

All *EdgeEye* elements share a similar structure. Aided by the GStreamer plugin architecture, we can easily integrate DNNs into GStreamer pipelines for different tasks by developing *EdgeEye* elements for them.

EdgeEye API

Through the JSON-RPC based API, applications can control both the pipeline and the individual *EdgeEye* elements. The API is task-specific. For example, the interface for object detection requires an input parameter to specify the DNN model used by the *EdgeEye* element, and the output is the detection results including bounding boxes and confidence level of the detection. The interface is independent of the underlying implementation of *EdgeEye* elements. Application developers does not need to care about the deep learning framework (Caffe, TensorFlow, etc.) and model (DetectNet, YOLO, SSD, etc.) used to implement the object detection function. *EdgeEye* provides a Javascript wrapper for those interfaces. Javascript is widely used to develop IoT applications, even for resource-constrained devices [117, 118]. The following code snippet shows how to use the Javascript wrapper to configure a DetectNet element to detect dogs. We ignore the code to load the deep learning model file and construct the pipeline. The application creates a DetectNet element instance through the handle of the pipeline. If succeeds, it sets an event handler to receive detection results.

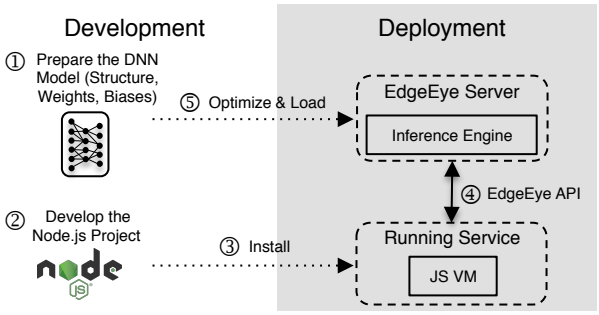


Figure 4.5: An example service powered by *EdgeEye*.

```
pipeline.create('DetectNet', {
  model: dogDetectorModel
}).then(element => {
  element.on('objectDetected', event => {
    // work with the detection result
    // (event.boundingBoxes, event.confidence)
  });
});
```

Work with *EdgeEye*

We can leverage *EdgeEye* from either edge services or standalone devices. This chapter only discusses edge services. Fig. 4.5 illustrates the steps to develop an *EdgeEye* powered edge service. First, we need to design and train the deep learning model with the deep learning framework we choose. Next, as usual, we develop the edge service logic with Javascript. The Javascript code will be combined with the trained deep learning model and other assets to build a deployable edge service. In the deployment phase, users install the service into the edge computing platform, which uses the *EdgeEye* API to upload the model to the *EdgeEye* server and starts the execution of the deep learning inference engine.

4.3 Implementation

We have built the prototype of *EdgeEye* on a desktop with Ubuntu 16.04 LTS. The desktop has an Intel i7-6700 CPU @ 3.40GHz, a Nvidia GeForce GTX 1060 6GB, and 24GB system RAM. We installed TensorRT 3.0.4 and CUDA 9.0 on the desktop.

We used Kurento [119] to implement the GStreamer pipeline management and JSON-RPC interface. Kurento is an open source WebRTC [120] media server. It provides the tools to generate the stubs on both server and client side for the JSON-RPC based API. Although its main goal is to support WebRTC related features and it provides many unrelated features for *EdgeEye*, its modular design can be reused to implement an edge computing framework. We used Node.js to implement the model repository and the HTTP interface.

We have implemented the DetectNet element for object detection with C/C++ based on TensorRT and CUDA and plan to build the SSD and YOLO elements in the future. The current prototype only uses Nvidia's GPU inference engine, but we will develop elements using Intel's inference engines as well. We also plan to develop elements for other purposes, such as semantic segmentation and activity detection.

TensorRT provides several optimizations for the GPU inference engine including layer & tensor fusion, FP16 and INT8 precision calibration, and kernel auto-tuning. These optimizations improve the inference performance significantly. Therefore, by leveraging these optimizations in TensorRT, *EdgeEye* can achieve higher inference performance than Caffe with the same DNN model. We evaluated the performance of the DetectNet element with a dog detector model. We used two video files with the same content (dogs), encoding (H.264) and length (2 minutes, 29.97 frames per second), but different resolutions (1280x720 and 640x360 respectively). We ran each test 10 times and averaged the results. For comparison purpose, we also tested Caffe (20180403-snapshot from Github [121]) with the same DNN model to get the average forward pass time. The forward pass time is the time taken for a batch of data to flow from the input layer of the network to the output layer of the network. Based on the average forward pass time, we calculated the FPS (Frames

Table 4.1: Inference speed test results

Configuration	Mean FPS
1. *EdgeEye* (video: 1280x720)	**55**
2. *EdgeEye* (video: 640x360)	**76**
3. Caffe (GPU, CUDA 9.0)	54
4. Caffe (CPU, w/o optimization)	0.49
5. Caffe (CPU, w/ optimization)	8.6

Per Second) values for Caffe framework. These values are the theoretical upper bound performance we can achieve with Caffe, because we omit overheads related to video decoding, buffer movement, and video scaling in the test. Three Caffe configurations were used in the test: i) the master branch of Caffe executing with GPU, ii) the master branch of Caffe executing without GPU and no optimization for CPU, iii) the Intel branch of Caffe executing with optimizations for Intel CPUs. We used batch size = 1 for all the tests because we want low latency.

Table 4.1 shows the results. We can see *EdgeEye* gets the highest FPS because of the optimizations in TensorRT, even though it needs to do a lot of extra work. In conclusion, GPU acceleration is very important to get high-performance inference, and the optimized inference engine (TensorRT) can further boost the performance. Inference accuracy data is not provided here because we use Nvidia's optimizer for TensorRT directly. Nvidia claims their GPUs can deliver massive performance improvements with the near-zero loss in accuracy [122].

4.4 Example Application

In this section, we introduce the design and implementation of an *EdgeEye* powered application — *home monitor*. Home monitor was built for the *ParaDrop* platform [8]. It can also be deployed in other edge computing platforms with minor modifications.

As shown in Fig. 4.6, the application has three major components: an edge service deployed in a *ParaDrop* AP, a cloud server, and a mobile app.

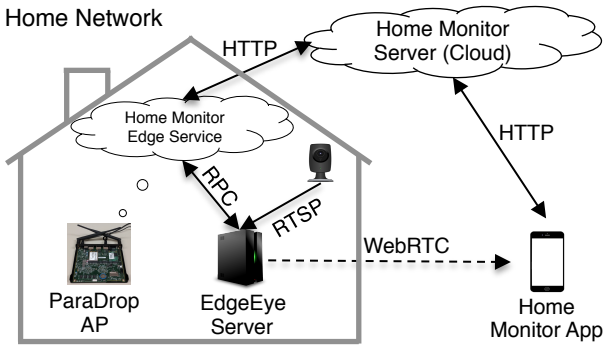


Figure 4.6: An example application built with *EdgeEye*.

The edge service uses the *EdgeEye* API to orchestrate the video analysis tasks and receive analysis results. The video stream is received from an IP camera with RTSP protocol support. The edge service also connects to the cloud server for WebRTC signaling. With the WebRTC support, users can define rules that edge service follows to set up a P2P connection to the mobile app and stream real-time video as well as analysis results to users with minimum latency.

The most important responsibility of the cloud server is WebRTC signaling. Other responsibilities include account management and rules management.

We developed the mobile app with the React-native framework. With the mobile app, we can receive real-time notification from the edge service, and view the real-time video stream from cameras remotely. Users can also specify some rules about the detection results.

Because of firewalls, NATs, and other middle-boxes on the Internet, a STUN/-TURN server is required to enable the Interactive Connectivity Establishment (ICE) for WebRTC connections. We installed an open source STUN/TURN server — coturn [123], in a cloud server with a public IP address.

The home monitor application currently only supports object detection. Based on the rules defined by users, it can record the video or send a notification to users, and then stream the live video to users if they want to. With other *EdgeEye* elements, we can easily support more use cases. For example, we can develop a face verification element to support a smart lock application. With *EdgeEye* framework,

video streams do not need to leave the home network for analysis. This approach saves bandwidth, provides reliable and low latency service, and also enables high privacy protection.

4.5 Discussion

Chip vendors, IP vendors, tech giants, and startups are building and shipping specialized processors for deep learning applications [124]. Some companies are also building products equipped with these kinds of specialized processors. For example, at the AWS re:Invent global summit 2017, Amazon announced DeepLens, a deep learning enabled wireless video camera for developers [125]. Google Clips is another wireless smart camera with machine learning enabled [126]. The advance in specialized processors can help *EdgeEye* to improve its performance. Unlike Amazon DeepLens and Google Clips, *EdgeEye*'s resources are shared by multiple smart devices. In addition, *EdgeEye*'s flexibility and multi-tenancy enable developers to try new algorithms easily at the edge.

The development of *EdgeEye* is still ongoing. Current work focuses on Convolutional Neural Networks (CNNs). We plan to support activities analysis in video streams through Recurrent Neural Networks (RNNs).

For now, we only work on computer vision applications in the smart home scenario, but *EdgeEye*'s architecture is also useful for other scenarios, such as city deployment and workplaces.

4.6 Conclusion

In this chapter, we present *EdgeEye*, a flexible, extensible and efficient edge computing framework to develop real-time video analytics applications. *EdgeEye* has a modular design based on GStreamer media framework, and it provides tools to deploy and execute DNN models in an easy and efficient way. By providing a high-level API, *EdgeEye* simplifies the development and deployment of deep

learning based video analytics applications. We discuss the motivations to design such a framework, and introduce its implementation. We also present an example application built with the *EdgeEye* framework to show its capability and extensibility. *EdgeEye* is an ongoing work, we are working on more elements for different computer vision tasks based on different inference engines.

5 Enabling Lightweight Multi-tenancy at the Extreme Edge

5.1 Introduction

After introducing the applications leveraging the power of edge computing and our efforts to simplify the process to develop applications requiring low latency offloading, we introduce our work on edge computing infrastructure in this chapter. We believe the availability of easy-to-use edge computing platforms is crucial for the adoption of edge computing.

This chapter presents a specific edge computing framework, *ParaDrop*, implemented on Wi-Fi Access Points (APs) or other wireless gateways (such as set-top boxes), which allows third-party developers to bring computational functions right into homes and enterprises. The Wi-Fi AP is ubiquitous in homes and enterprises, is always "on" and available, and sits directly in the data path between the Internet and end-user devices. For these reasons, the Wi-Fi AP was selected for *ParaDrop* as the edge computing platform. *ParaDrop* uses a lightweight virtualization framework through which third-party developers can create, deploy, and revoke their services in different APs. Their services are inside containers that allow them to retain user state and also move with the users as they change their points of attachment. *ParaDrop* also recognizes that Wi-Fi APs are likely to be resource limited for many different types of applications, and hence allows for tight resource control through

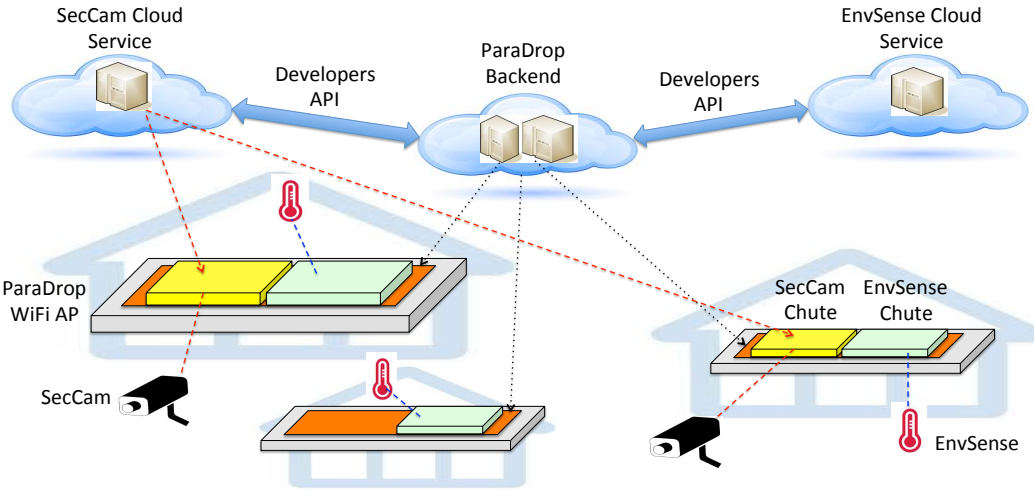


Figure 5.1: Overview of *ParaDrop*. There are three components in the system: the *ParaDrop* backend, *ParaDrop* gateways, and the developer API. Developers can deploy services in containers (we call them chutes in *ParaDrop*, as in parachuting a service on-demand) to the gateways through the backend server. The chutes are deployed using the developer API. Two different chutes are illustrated in this figure: 1) *SecCam*: a wireless security camera service that collects video from an in-range security camera and does some local processing to detect motions; and 2) *EnvSense*: a system deployed in buildings to monitor temperature and humidity with sensors. We illustrate these two applications in Section 5.5.

a managed policy design.[1]

The *ParaDrop* framework has three key components: (i) a virtualization substrate in the Wi-Fi APs that hosts third-party computations in isolated containers (which we call *chutes*); (ii) a cloud backend through which all of the *ParaDrop* APs and the third-party containers are dynamically installed, instantiated and revoked; and (iii) a developer API through which developers can manage the resources of the platform and monitor the running status of APs and chutes. Compared to other heavyweight counterparts, *ParaDrop* uses more efficient virtualization technology based on Linux containers [127] rather than the Virtual Machines(VMs),

[1]Wi-Fi APs, like all other compute platforms, will continue to get faster and powerful, and the capabilities installable in applications will continue to improve. Our current low-end hardware can perform image processing and motion detection in video feeds.

meaning we can provide more resources for services with the given hardware. The virtualization substrate has evolved from LXC [128] to one based on Docker [129]. The change to Docker was due to easy-to-use tools and wider adoption; this reduces the complexity for new developers to create services on a platform they may be unfamiliar with. We also developed a backend server to manage the gateways through a real-time messaging protocol, Web Application Messaging Protocol (WAMP) [130], which guarantees very low latency in service deployment and monitoring.

We demonstrate the capabilities of this platform by showing useful third-party applications utilizing the *ParaDrop* framework. Fig. 5.1 gives an overview of the architecture of *ParaDrop* and shows two example applications. The first example application is a security camera application that connects with wireless cameras in-range to detect motions. The second application connects to environment sensors in the home, such as temperature and humidity, to determine the position of actuator controlled heating or cooling vents. In addition to the low-latency advantages of *ParaDrop*, the platform also has unique privacy advantages. A third-party developer can create a *ParaDrop* service that allows sensitive user data (video camera feed) to reside locally in the *ParaDrop* AP and not send a continuous feed over the open Internet. Over the last several years, we have installed and deployed *ParaDrop* as a platform using many different use cases. We can also build edge services for the *ParaDrop* platform with edge offloading frameworks. For example, in section 4.4, we describe a service deployed in the *ParaDrop* platform with the *EdgeEye* framework.

5.2 *ParaDrop* Overview

A decade or two ago, the desktop computer was the only reliable computing platform in the home where third-party applications could reliably and persistently run. However, diverse mobile devices, such as smartphones and tablets, have deprecated the desktop computer. Today, persistent third-party applications are often run on remote cloud-based servers. While cloud-based third-party services have many advantages, the rise of edge computing stems from the observation

that many services can benefit from a persistent computing platform in end-user premises.

With end-user devices going mobile, there is one remaining device that provides the capabilities developers require for their services, as well as, the proximity expected from an edge computing framework. The gateway — which could be a home Wi-Fi AP or a cable set-top box provided by a network operator — is a platform that is continuously on, and, due to its pervasiveness, is a primary entry point into the end-user premises for such third-party services.

We want to push computation onto the home gateways (e.g., Wi-Fi APs and cable set-top boxes) for following reasons:

- The home gateways can handle it. Modern home gateways are much more powerful than what they need to be for their networking workload. Furthermore, if one is not running a web server out of the home, the gateway sits dormant a majority of the time (e.g., when no one in the home is using it).

- Utilizing computational resources in the home gateway gives us a local footprint within the home for end-devices that are otherwise starved for computational resources. These include sensors such as many inexpensive Internet of Things (IoT) sensors and actuator components. Using *ParaDrop*, developers can offload some computation of one (or many) IoT devices onto the AP without the need for cloud services or a dedicated desktop. In particular, if one installs different sensors from different vendors, this advantage becomes even more apparent. If a home has controls for blinds, temperature sensors, a specific thermostat, etc., the computational function to make decisions on when to actuate some of these devices are best made, locally, in the *ParaDrop* AP. They can achieve better efficiency by eliminating unnecessary network traffic, which has additional benefits to avoid network congestion when the network traffic is high.

- Pervasive hardware: Our world is quickly moving towards households only having mobile devices (tablets and laptops) in the home that are not always

on or always connected. Developers can no longer rely on pushing software into the home without also developing their own hardware too.

A developer-centric framework

A focus on edge computation would require developers to think differently about their application development process. However, we believe there are many benefits to a distributed platform such as *ParaDrop*. The developer has remained our focus in the design and implementation of our platform. Thus, we have implemented *ParaDrop* to include a fully featured API for development, with a focus on a centrally managed framework. Through virtualization, *ParaDrop* enables each developer access to resources in a way as to completely isolate all services on the gateway. A tightly-controlled resource policy has been developed, which allows fair performance between all services. Users (owners or administrators of *ParaDrop* gateways) can manage and monitor *ParaDrop* gateways with centralized tools similar to the dashboards for cloud computing platforms. We strive for *ParaDrop* to be easily deployed and managed by administrators. At the same time, it should be able to improve the user's experience transparently or make specific tasks easy.

ParaDrop capabilities

ParaDrop takes advantage of the fact that resources of a gateway are underutilized most of the time. Each service, referred to as a *chute*, can borrow CPU time, unused memory, and extra disk space from the gateway. This allows vendors an unexplored opportunity to provide added value to their services exploiting the proximity footprint of the gateway.

Fig. 5.2 shows a *ParaDrop* gateway, along with two services to motivate our platform: *SecCam* and *EnvSense*. The current instance of the *ParaDrop* gateway has been implemented on a PCEngines single board computer [131] running Snappy Ubuntu on an AMD APU 1GHz processor with 2GB of RAM. By default, the gateway has a 16GB SD card for storage, with the option to use a larger SD card if necessary. And we have used 64GB SSD (solid-state drive) on the gateway to

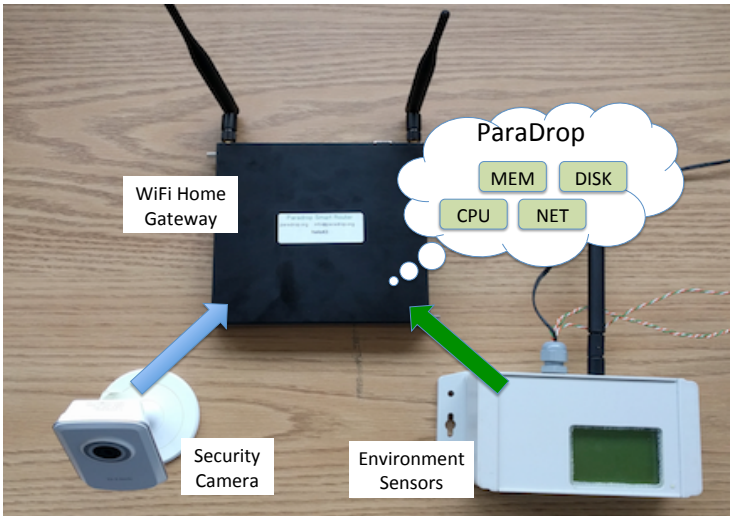


Figure 5.2: The fully implemented *ParaDrop* gateway, which shares its resources with two wireless devices including a Security Camera and an Environmental Sensor.

support services requiring large storage space. This low-end hardware platform was chosen to showcase *ParaDrop*'s capabilities with existing gateway hardware. We have implemented two third-party services by migrating their core functionality to the *ParaDrop* platform to demonstrate its potential. Each of these services contains a fully implemented set of applications to capture, process, store, and visualize the data from their wireless sensors within a virtually isolated environment. The first service is a wireless security camera based on a commercially available D-Link 931L webcam, which we call *SecCam*. The second service is a wireless environmental sensor designed as part of the *Emonix* research platform [132], which we refer to as *EnvSense*. Leveraging the *ParaDrop* platform, the two services allow us to motivate the following advantages of *ParaDrop*. While some of them generally apply to edge computing platforms, some of the advantages stem from our construction and use of Wi-Fi APs as the hosting substrate. They include:

- *Privacy:* Many sensors and even webcams today rely on the cloud as the only storage mechanism for generated data. By leveraging the *ParaDrop* platform, the end-user no longer must rely on cloud storage for the data generated

by their private devices, but can instead borrow disk space available in the gateway for such data.

- *Low latency:* Many simple processing tasks required by sensors are performed in the cloud today. By moving these simple processing tasks onto gateway hardware which is one hop away from the sensor itself, a reliable low-latency service can be implemented by the developer.

- *Proprietary friendly:* From a developer's perspective, the cloud is the best option to deploy their proprietary software because it is under their complete control. Using *ParaDrop*, a developer can package up the same software binaries and deploy them within the gateway to execute in a virtualized environment, which is still under their complete control.

- *Local networking:* In the typical service implemented by a developer, the data is consumed only by the end-user, yet stored in the cloud. This requires data generated by a security camera in the home to travel out to a server somewhere on the Internet and upon the end-user's request travel back from this server into the end-user's device for viewing. Utilizing the *ParaDrop* platform, a developer can ensure that only data requested by the end-user is transmitted through Internet paths to the end-user's device.

- *Additional wireless context:* A Wi-Fi AP can sense more information about end-devices, such as, the proximity of different devices to each other, location in specific rooms and much more. If such an API is exposed to developers (with proper privacy management), it enables new capabilities complementary to other means of accessing the same information.

- *Internet disconnectivity:* Finally, as services become more heterogeneous, they will move away from simple "nice to have" features, into mission-critical, life-saving services. While generally accepted as unlikely, a disconnection from the Internet makes a cloud-based sensor completely useless and is unacceptable for services such as health monitoring. In this case, a developer could leverage

the always-on nature of the gateway to process data from these sensors, even when the Internet seems to be down.

5.3 System Design

In this section, we first discuss the challenges to design the *ParaDrop* system and explain our solutions to overcome them. We then give an overview of the system architecture and explain the key features to support deploying services on the *ParaDrop* platform.

Challenges and Solutions

ParaDrop uses virtualization technology to provide an isolated environment to services running on the gateway. One key difference of the *ParaDrop* platform compared to cloud computing platforms is the gateway hardware has lower performance than servers in datacenters. As a result, the efficiency of the virtualization technology is the most important consideration in the system design. There are two types of virtualization approaches: Virtual Machines(VMs) and containers. Hypervisor-based VMs virtualize at the hardware layer, while containers virtualize at the operating system layer. Felter et al. did a measurement study to compare the overhead of VMs and Linux containers. More specifically, they compared KVM [133] with Linux containers [127]. They concluded that KVM's complexity makes it not suitable for a workload that is latency-sensitive or has high I/O rates, though both KVM and container have very low overhead on CPU and memory. The VM-based approach provides a fully virtualized environment, which means developers have high flexibility to choose operating systems. However, since it provides access to hardware only, we need to install an operating system in the virtual environment, which quickly gobbles up resources on the gateway, such as RAM, CPU, and disk. Ha et al. proposed an optimized dynamic VM synthesis approach which makes VMs less heavyweight than they appear to be [134]. However, their approach used a very high-performance server as the cloudlet. In

contrast to VMs, a container is more lightweight and faster to boot because it does not run a full OS on the virtual hardware. Its high efficiency and low footprint are critical for a low-performance hardware platform. Moreover, we do not think the restriction to use the Linux operating system would be a big concern for developers. Therefore, we choose a container-based virtualization rather than VMs to provide an isolated environment for the services deployed on the gateways. Several management tools are available for Linux containers, including LXC [128], Docker [11], etc.. Due to its feature set and ease of use, Docker has rapidly become the standard management tool and image format for containers [135]. A unique feature of Docker is layered filesystem images, usually implemented by AUFS (Another UnionFS) [136]. The layered filesystem provides the opportunity for us to reduce space usage for the images and simplify the file system management. So we chose Docker to manage the containers in the *ParaDrop* gateways in the latest version of *ParaDrop* implementation.

Another challenge encountered while designing *ParaDrop* is that developers normally do not have direct access to the gateways in the deployment because of NATs or firewalls. Therefore, it is hard for them to manage and debug the services running on the gateways. The Virtual Private Network (VPN) is one approach to overcome that challenge. However, with a VPN, the backend server needs to maintain the connections to all the gateways. Developers also need to configure VPN on their machines in order to access the gateways. To simplify the backend server implementation and developers' work, we use the Web Application Messaging Protocol (WAMP) to enable bi-directional communication channels between the *ParaDrop* backend and gateways. WAMP is an open standard WebSocket subprotocol that provides two application messaging patterns in one unified protocol: Remote Procedure Calls + Publish/Subscribe [130]. With WAMP, we can transmit time-sensitive messages between backend server, developer tools, and gateways with very low latency.

Unlike servers in datacenters, it is hard for us to control the hardware and upgrade software on it after we deploy *ParaDrop* gateways in the network edge — user's home or office. We need a software system with secure and transactional

update capability. The software system also needs to support image backup and rollback features so that we can manage the software running on gateways easily and flexibly. We built the software stack for the gateway based on Ubuntu Core [137] which has met our requirements.

ParaDrop System Architecture Overview

ParaDrop has three core components as shown in Fig. 5.1. The backend and gateways are connected by a WAMP message router. The gateways provide the virtualized environment for the services. Whereas the backend maintains the information of the gateways and users. It also provides a repository (*ChuteStore*) to store files for the services which can be deployed on the gateways. It manages the resource provision to developers and the services running on the routers in a secure way.

Implementing Services for the *ParaDrop* Platform

A service is deployed on the *ParaDrop* platform in a package called a *chute* (short for parachute). A user can deploy many chutes (Fig. 5.3) to an AP, thanks to the low-overhead container technology. These chutes allow for isolated use of computational resources on the gateway. As we design and implement services on gateways, we can, and should, separate these services into unique chutes.

There are several primary concerns of the *ParaDrop* platform including installation procedure, API, networking configuration, and resource policies.

Dynamic installation: In order for end-users to easily add services to their gateway, each service should have the ability to be dynamically installed. This process is possible through the virtualization environment of each chute. When an end-user wishes to add a service to their home, they simply register an account to *ParaDrop*. Using the *ParaDrop* API, the user links the account with their gateways. If a service utilizes a wireless device, the gateway can fully integrate with the service without any interference from the end-user.

***ParaDrop* API:** The focus of *ParaDrop* is providing high-quality services to end users via third party developers. A seamless RESTful API enables developers to

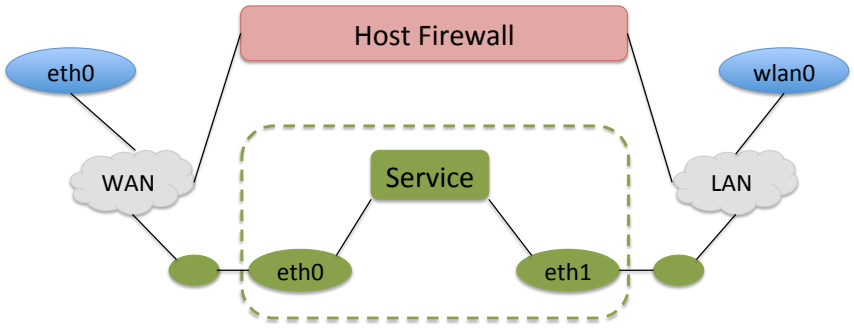


Figure 5.3: A chute is running on an AP. The dashed box shows the block diagram representation of a chute installed on a *ParaDrop* gateway. Each chute hosts a stand-alone service and has its own network subnet.

have complete control over the configuration of their chutes. As services evolve, the API will provide all the capabilities required without the need for modification to the configuration software. This is possible through the use of a JSON-based data backend which allows abstract configuration and control over each chute.

Network setup: The networking topology of a dynamic, virtualized environment controlled by several entities is very complex. In order to maintain control over the networking aspects of the gateway, we leveraged an SDN (Software-defined Networking) paradigm. All configuration related to networking between the chutes and the gateway are handled through a cloud service, which is interfaced by the developers and network operators.

Resource policies: The multi-tenancy aspects of *ParaDrop* require tight policy control over the gateway and its limited resources. Currently, the major resources controlled by *ParaDrop* include CPU, memory, and networking. Using the API, developers specify the type of resources they require depending on the services they implement. Through the management interface, the network operator can dynamically adjust the resources provided to each chute. These resources are adjusted first by a request sent to the chute, and if not acted upon, then by force through the virtualization framework tools.

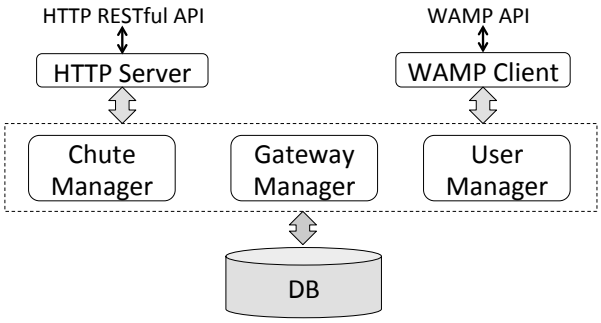


Figure 5.4: The architecture of the *ParaDrop* backend. The backend manages all the resources of the *ParaDrop* platform and provides APIs for users to deploy services on the gateways.

5.4 Implementation

In this section, we introduce the implementation details of the *ParaDrop* platform. Though the design motivations and strategies are maintained, *ParaDrop* has evolved from a VM-based version, to a Linux containers + LXC [127] based architecture, and finally to the current Linux container + Docker [129] based architecture. We only cover the implementation of the latest version of *ParaDrop* in this chapter. As we introduced in Section 5.3, WAMP is used to connect the *ParaDrop* backend and the gateways. WAMP's two messaging patterns exactly match our requirements to manage and monitor the *ParaDrop* gateways with minimum latency. We built our system based on Autobahn [138], which provides open source implementations of the WebSocket Protocol and WAMP. We chose crossbar.io as the WAMP router for the *ParaDrop* platform. Crossbar.io [139] is an open source multi-protocol application router based on Autobahn and WAMP. We installed crossbar.io on an Ubuntu 14.04 machine which is accessible from both the backend server and the gateways.

The *ParaDrop* Backend Implementation

The *ParaDrop* backend manages the platform resources in a centralized manner. Initially, we implemented the *ParaDrop* backend with the Python programming

language based on the Twisted [140] network framework. In the latest version, we used Node.js to build the *ParaDrop* backend due to the event-driven architecture of Node.js. The major components of the backend are shown in Fig. 5.4. Two important interfaces are implemented for the backend server:

- The WAMP API for the *ParaDrop* gateways. The backend communicates with all the gateways to dispatch commands, receive responses, and receive status reports. The soft real-time characteristic of WAMP guarantees the minimal latency in the message exchanging [130].
- The HTTP RESTful API for developers, end-users, administrators, and gateways. On the user side, the backend server aggregates the information from all the gateways and provides the information to the *ParaDrop* frontend. Users can then view the information in a graphical format from the server. The backend server also stores some persistent information about the *ParaDrop* platform deployment, e.g., the location and configuration of the gateways. Moreover, the backend server relays the commands from users to gateways. Such as starting a chute on one or more specific gateways, and then relaying the responses from the gateways to users.

In addition to the above interfaces, the backend server also contains several authentication mechanisms. To get permissions for a resource, all users need to register for an account on the backend server. The backend server stores information about the users, gateways, and chutes in a MongoDB database [141]. We have implemented a repository to manage the published chutes submitted to the backend server. Users are able to deploy chutes from that repository to their gateways through the web frontend.

The *ParaDrop* Gateway Implementation

The *ParaDrop* gateway is the execution engine of the *ParaDrop* platform. Fig. 5.5 shows the major software components on a *ParaDrop* gateway. First of all, we need to securely and reliably maintain the *ParaDrop* software components on the

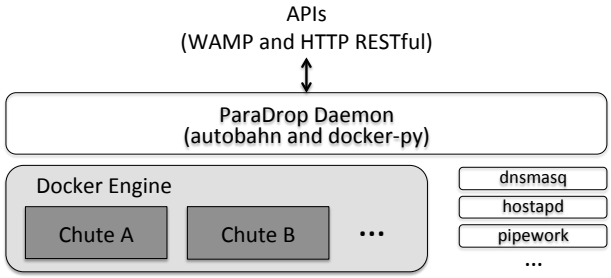


Figure 5.5: Major software components on a *ParaDrop* gateway. All the components are snappy packages. We implemented the *ParaDrop* daemon with Python from scratch. And we made the snappy packages for dnsmasq and hostapd based on the corresponding open source projects. The Docker engine is packaged by Ubuntu. All the snappy packages can be securely and transactionally updated over-the-air if required.

operating system of a gateway. In Ubuntu Core, every software component is a Snappy package, which allows transactional upgrades or rollbacks of the component reliably [137]. To that end, we chose Ubuntu Core as the operating system.

On the one hand, we want to provide an isolated and virtualized environment for the chutes deployed on a gateway. On the other hand, we want to keep the virtualization overhead as low as possible because the hardware platform of the gateway is not as powerful as a server in cloud computing platforms. These are just a couple of reasons why we chose Linux containers and Docker to build the virtualized environment for the chutes. Docker has features such as layered image and NAT, which ease the development, management, and deployment of chutes. Docker also effectively solves the dependency problem when developers migrate software tested in a development environment to a deployment environment. Since Docker is an App-container, it is perfect for a chute that only requires one process. If a complex service needs to run multiple processes, we can either launch multiple containers or use the Docker supervisor. Also, since Docker is a pure execution environment, we need to manage the networking environment in our implementation.

A *ParaDrop* daemon (also called Instance tools) runs on the gateway to im-

plement all the functions required by the *ParaDrop* platform. It implements all Docker-related features based on a Python wrapper of Docker APIs. Additionally, it exposes the controlling and monitoring interfaces to outside world. Its major responsibilities include:

- Registers the gateway to the *ParaDrop* backend.
- Monitors the gateway's status and reports it to the *ParaDrop* backend.
- Manages resources and processes including virtual wireless devices, firewall, DHCP, hostapd, etc., to provide an environment for the chutes managed by the Docker engine and also for the devices connecting to the gateway.
- Receives RPCs and messages from the *ParaDrop* backend and manages containers on the gateway accordingly, e.g., install, start, stop, and uninstall chutes.

The *ParaDrop* daemon interfaces support both WAMP and HTTP protocols. Therefore, a *ParaDrop* gateway can be accessed by the developer tools indirectly through the WAMP message router or directly if they are on the same network. Every chute must define the resource requirements in a config file, as the *ParaDrop* daemon enforces resource policies on all installed chutes. These policies are enforced when it is creating or starting a chute instance. *ParaDrop* supports enforcement for the resources listed below:

- CPU: CPU resource for a chute is controlled by a *share* value of the container. We can specify the share value when we create a container, or change it on-the-fly when a container is running. The *share* value defines relative share of the CPU resource that one chute can use when it competes with other chutes. It does not limit the maximum CPU resource a chute can use. We explain that in detail in Section 5.6.
- Memory: The maximum memory size can be used by a chute is restricted by a value defined in its config file.

- Network: The *ParaDrop* daemon tracks all the network interfaces used by chutes and restricts their bandwidth by traffic shaping. We may implement a strategy based on *share* values similar to the CPU resource management in a future release.

Some edge computing applications require the storage resource to cache content, or to provide local storage services to users. Hence, we need a flexible way to manage the limited storage resources in the gateways. We plan to implement policies to manage the block devices based on the Linux kernel's device mapper and Docker's *devicemapper* storage driver [142]. Example policies include disk size quota for a chute, read/write speed. Currently, we only define a common maximum disk size (1GB) for all chutes.

The *ParaDrop* Developer Console Implementation

We implemented the developer console with Python. Since a chute can be implemented with any languages or libraries, the developer console is agnostic to the contents of chutes. Essentially a chute is a Docker image with *ParaDrop*-specific files. The developer console provides the tools that we can use to interact with the *ParaDrop* platform. Developers can create chutes locally, upload chutes to the backend and install chutes from local machine or backend to the gateways if they have permissions to do so. We can also monitor the status of the chutes and gateways. If a developer has direct access to the gateways (in development environment), he or she can access the gateways directly without the WAMP router.

The *ParaDrop* Web Frontend Implementation

The *ParaDrop* platform has a Web frontend. It provides all the features of the developer console except building a chute. Moreover, it provides a better user interface for users and administrators to install chutes on their gateways in an intuitive way. The web interface to monitor the status of chutes and gateways is also nicer than the command line tools.

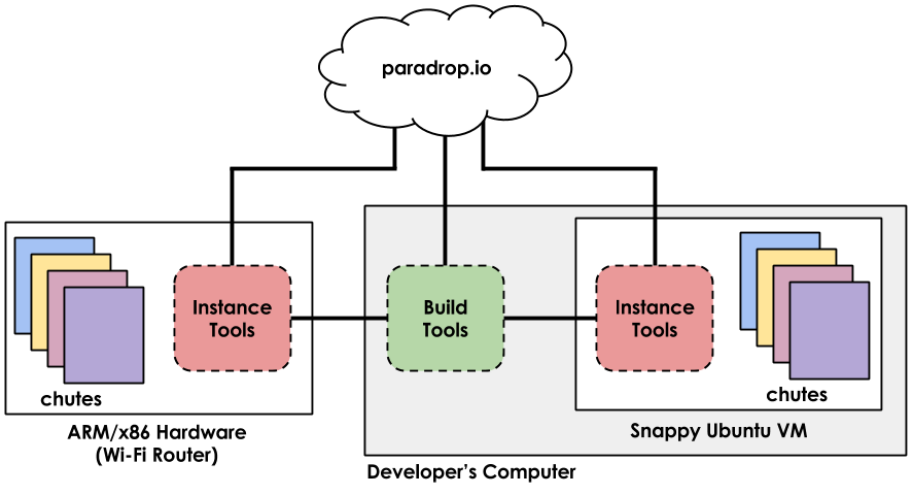


Figure 5.6: The *ParaDrop* platform from a developer's view: We refer to Build Tools when we talk about the command line program running on developer's development computer that control and communicates with the rest of the *ParaDrop* platform. The Instance Tools (including the *ParaDrop* daemon) leverage Docker to allow *ParaDrop* apps to run on router hardware. This "hardware" could be a *ParaDrop* gateway, a Raspberry Pi, or even a virtual machine on your computer that acts as a router (which is why we call it an Instance sometimes).

The *ParaDrop* Workflow

To develop an application for the *ParaDrop* platform, a developer needs to interact with two components:

- The build tools in the developer's computer.
- The instance tools in the *ParaDrop* daemon.

Fig. 5.6 shows the two components that a developer needs to work with. By providing tools to create and manage chutes, the platform implementation is transparent to developers. They can view the *ParaDrop* hardware as familiar resources close to the user's mobile devices and focus on the application logic development.

It is recommended a developer test the application locally, and then pack the binary with the configuration files into a chute with the build tools. Upon successful

chute packing, testing with the local *ParaDrop* gateway should follow. Once all the bugs and functional verification tests have been completed, the developer can either install the chute to the routers or publish the chute to the *ChuteStore*.

5.5 *ParaDrop* Applications

Developers can deploy various chutes (applications and services) on the *ParaDrop* platform. Any service currently running on a cloud computing platform that can take advantage of the edge computing resources offered by the *ParaDrop* platform can be easily ported with just a few changes. In order to make the service compatible with the *ParaDrop* platform, we need to provide some configuration files to define the resource requirement of the service, e.g., CPU, memory, network. With the *ParaDrop* developer tools, we can package the binary files, scripts, Dockerfile, and *ParaDrop* configuration files to a chute. The chute can then be deployed on *ParaDrop* gateways to provide the service. Fig. 5.7 illustrates the process to deploy and launch a chute on a gateway. Depending on the applications requirements, we can either deploy the whole service to the *ParaDrop* platform; or for a complex service composed of many microservices, we can deploy parts of the service (some microservices) to the *ParaDrop* platform. In the latter case, some microservices running in the cloud collaborate with the microservices running on the *ParaDrop* platform to provide the service to end-users.

In this chapter, we present two IoT example applications in detail. IoT is becoming a huge part of the networking world. Yet, many IoT devices rely on backend services that must traverse the Internet to utilize their full potential. Using *ParaDrop*, we can pull that intelligence into the edge — the gateways.

The *SecCam* Service

In this section, we present a walkthrough about using a wireless video camera with a *ParaDrop* gateway to implement a security camera service called *SecCam*. The

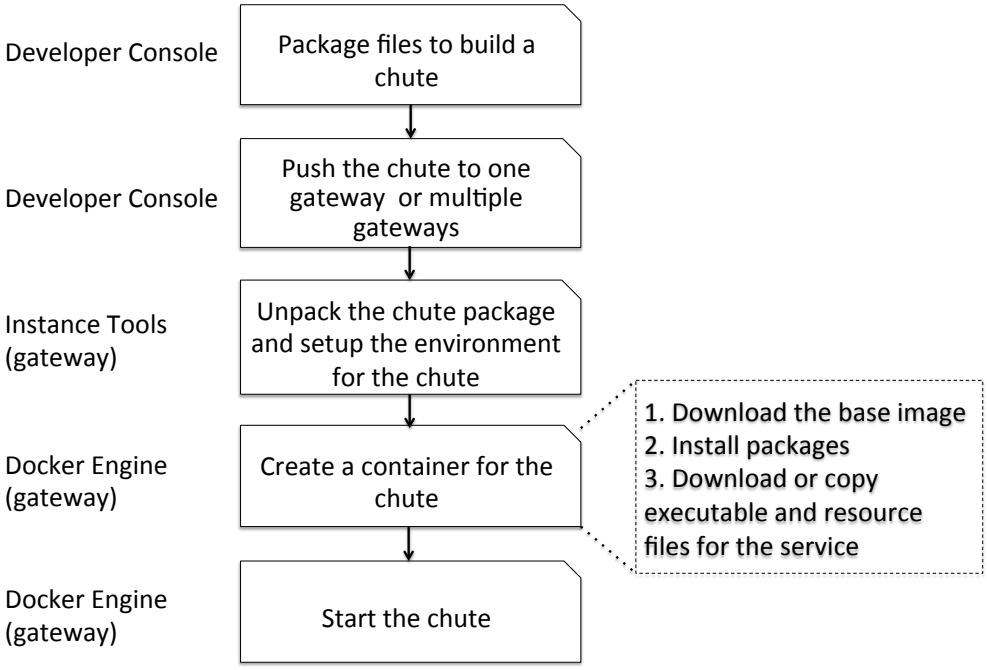


Figure 5.7: The process to build, deploy and launch a chute on a gateway. The step to create the container for the chute could be simplified by building the Docker images and uploading them to a private Docker repository beforehand.

SecCam service is based on a commercially available wireless IP camera (D-Link 931L webcam), where we took the role of developer to fully implement the service.

D-Link provides a cloud service "mydlink" [143] to which users can register their camera and can access the camera remotely. They can view the image and modify the settings with a web client or mobile apps supporting "mydlink" cloud service. The cloud service is very convenient to setup, but the video data need to be pushed to the cloud platform and users do not have much control over it. We propose to move the functionality of the cloud service to the *ParaDrop* gateway to give users full control of the camera device and the video data.

For this service, we require network interfaces to communicate with the webcam and the Internet, as well as ample storage for images. Fig. 5.8 compares the camera service deployed in cloud and *ParaDrop*.

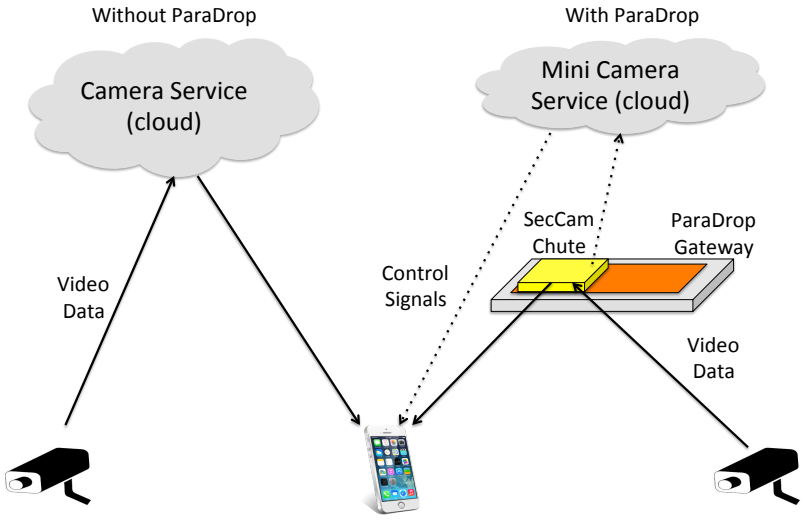


Figure 5.8: An IP camera service without *ParaDrop* and with *ParaDrop*. With *ParaDrop*, we can deploy the camera service into the gateway. If the mobile device has a direct connection with the camera service on the gateway, it will control the camera and get video data from it directly without relying on the service deployed in the cloud. If not, the camera service on the gateway still can do most work (e.g., motion detection) and it can create a peer-to-peer connection to the mobile device with the help of the mini camera service in the cloud. We do not discuss the mini camera service here for brevity. *ParaDrop* makes the chute deployment easy and flexible.

We need to create a chute for the *SecCam* service, which is responsible for:

- **Creating the *SecCam* SSID.** This SSID provides an isolated Wi-Fi network and subnet to the security cameras. The devices purchased by end-users do not have to be programmed when they arrive at the purchaser's home (they can be flashed with a default SSID and password by the company).

- **Image capture service.** This is a Python-based program with user-defined characteristics such as a threshold of motion, time of day, and rate of detection. The service captures images from an IP camera, calculates differences to detect motion, and stores those images to disk. The images stored on disk will then

```
owner: ParaDrop
date: 2016-02-06
name: SecCam
Description:
   This app launches a virtual wifi AP |
   and detects motion on a wifi webcam.
net:
   wifi:
      type: wifi
      intfName: wlan0
      ssid: seccamv2
      key: paradrop
      dhcp:
         lease: 12h
         start: 100
         limit: 50
resource:
   cpu: 1024
   memory: 128M
dockerfile:
   local: Dockerfile
```

Figure 5.9: The configuration file of a *SecCam* chute. It defines the environment of the container to run the chute.

```
FROM ubuntu:14.04
MAINTAINER ParaDrop Team <info@paradrop.io>

RUN apt-get update && apt-get install -y \
   apache2 \
   libapache2-mod-php5 \
   python-imaging

ADD http://paradrop.io/storage/seccam/srv.tar.gz /var/www/
RUN tar xzf /var/www/srv.tar.gz -C /var/www/html/
ADD http://paradrop.io/storage/seccam/cmd.sh /usr/local/bin
CMD ["/bin/bash","/usr/local/bin/cmd.sh"]
```

Figure 5.10: The Dockerfile of a *SecCam* chute. This Dockerfile is a simplified version. We need to download some files for the chute from a web server. In deployment, these files can be included in the chute package.

be visualized using a web server which runs inside the chute.

- **Web server.** The web server is the mechanism that allows the end-users on the local network to view live video, check the logs, and view the motion

detection images stored on the *ParaDrop* device.

A chute is described by a *ParaDrop* configuration file and a Dockerfile.

- The *ParaDrop* configuration file is a YAML [144] formatted file for *ParaDrop*. It describes the resource and environment requirements of the chute.
- The Dockerfile follows the specification of Docker. It defines the Docker image which is managed by the Docker engine on the *ParaDrop* gateways.

Fig. 5.9 shows the configuration file of the *SecCam* chute and Fig. 5.10 shows its Dockerfile. The Docker image is based on Ubuntu 14.04. The static file of the web server and the source code of image capture service will be installed on the image when we launch the chute on a *ParaDrop* gateway. Alternatively, we can store the pre-built image in a private Docker repository so that we can download and launch the chute directly.

The *EnvSense* Service

This service is a wireless environmental sensor designed as a part of the *Emonix* research platform [132]. Since the service is fully implemented, we only need to migrate the service to fit the *ParaDrop* platform, rather than rewrite it from scratch. The original service runs on a server to collect data from the sensors, process, store and visualized the data. After identifying the resources required to run the service, we can create a chute for it. The steps to create the configuration file are similar to the *SecCam* service so we omit them here for brevity.

We can divide the *EnvSense* service to multiple microservices. For example, we can run data collection and processing microservices on the *ParaDrop* platform and run data storage and visualization microservices on the cloud platform. It will be fairly easy with the development and management tools provided by *ParaDrop* to implement that change.

Other Possible Applications

Other than IoT, applications in other categories can also take advantage of the *ParaDrop* platform. For instance, Peer-to-Peer (P2P) technology is a popular approach to reducing the workload of the media server [145]. However, it is challenging to use P2P on mobile devices because of the following issues:

- Mobile devices are powered by batteries so that they can not afford the overhead to upload the media data or share the data with other peers.
- The wireless connections have restricted bandwidth compared to wired connections, in many cases, mobile devices do not have abundant bandwidth can be used to share media data with other peers.
- Compared to a wired network, a wireless network is dynamic, so the overhead to maintain the status of peers can be too high and offset the potential advantages of P2P technology.

A Wi-Fi gateway, as the last hop of mobile devices' connection to the media server, is a good place to deploy the P2P technology. With *ParaDrop*, they can be implemented in a chute in order to provide transparent service to the mobile devices.

Another type of application is a caching service. We built a media caching service on the *ParaDrop* platform to prefetch and cache video data from Netflix. That service can improve user experience by eliminating network congestions.

5.6 Evaluations

In this section, we discuss the evolution of the *ParaDrop* platform and evaluate the system regarding efficiency and effectiveness. We also discuss the system's usability and scalability.

Table 5.1: Overhead of the virtualization schemes on the first generation hardware platform of *ParaDrop* gateway

Test	Host	LXC	Lguest
Start Time (sec)	-	2	40
Packet Latency (ms)	0.04	0.20	39.0
Throughput Fairness	Host Chute	50% 50%	60%/40% 30%/30%

Efficiency of virtualization

The selection of a virtualization solution is the core of the *ParaDrop* platform. The platform itself has already been through three generations of virtualization schemes throughout its evolution.

- **Hypervisor-based virtualization: OpenWRT [146] + lguest.** In the first generation of the *ParaDrop* platform, we implemented the virtualization solution based on lguest. Lguest is a small x86 32-bit Linux hypervisor for running Linux under Linux [147]. A number of hypervisor solutions have appeared that use Linux as the core, including KVM [133] and lguest. We selected lguest for our first generation hardware platform, because of its relatively simple implementation (5000 lines of code) and availability in the mainline Linux kernel. Though the simplicity of lguest is attractive, it is slower than other hypervisors. A recent measurement study by Felter et al. confirmed that VMs have high latency in I/O operations [135] and they are not suitable to latency sensitive applications. Another problem of lguest is it is a paravirtualization hypervisor, which means the guest OSes need to be lguest-enabled. This increases the complexity when porting services and creates a barrier for developers to go through in adopting the platform.

- **Linux Containers: OpenWRT + LXC.** Looking for a more lightweight virtualization scheme, we migrated to the LXC (Linux container) management tool. Containers have lower overhead than hypervisor-based virtualization solutions, because they share the same operating system and do not emulate

hardware. There are two user-space implementations of Linux containers, each exploiting the same kernel features [128]. We selected LXC because it is flexible and has more userspace tools. In order to implement container-based virtualization, the application must be supported by the host Linux kernel. We did not think that would be a big concern for many applications. Dale et al. measured the overhead of the first two generations of virtualization schemes [148] and the results are shown in Table 5.1. Lguest needs much longer time to start because it needs to boot the guest operating system before it can run an application. It should be noted, the result regarding packet latency in Table 5.1 is not consistent with the results of KVM reported by Felter et al. [135]. Table 5.1 also shows the throughput fairness, LXC can achieve good fairness along with much lower overhead than lguest.

- **Linux Containers: Snappy Ubuntu + Docker.** Although the second generation virtualization software had very low overhead and the implementation on OpenWRT is quite efficient for the hardware, we began to realize the operating system disparity on the gateway and the cloud platform could be an obstacle to application development. Upon this realization, we migrated to Snappy Ubuntu and upgraded the hardware to a 64-bit processor platform. This was a significant change and caused many software architecture modifications in the process of going from not only an OS alteration, but also a virtualization modification. For the record, Docker has similar performance on start time, packet latency and throughput fairness as shown in Table 5.1 for LXC.

  To have a clear idea about the latency to deploy, start, stop and delete a chute on a gateway, we measured the time taken for these operations on the *ParaDrop* gateway with the PC Engines APU1 board. The results are shown in Table 5.2. We tested the system with the *SecCam* chute we described in Section 5.5. The test results depend on the network bandwidth because we need to download an Ubuntu 14.04 base image from the Docker repository, some Ubuntu packages from an Ubuntu repository and a package including

Table 5.2: Chute operations benchmark on the *ParaDrop* gateway with the PC Engines APU1 board

Operation	Time (sec)
Deploy	527
Start	5
Stop	17
Delete	7

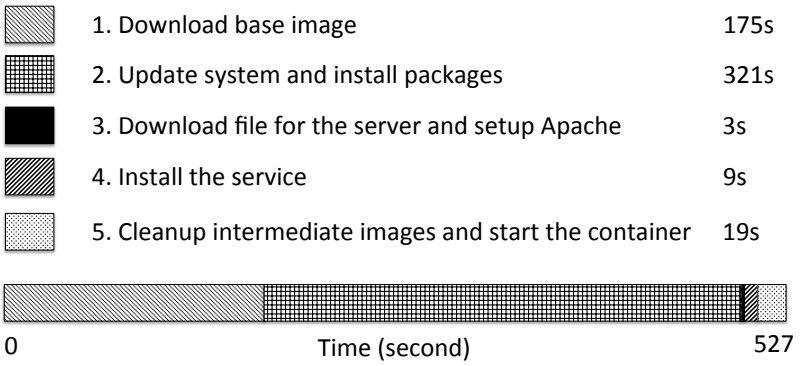


Figure 5.11: Time taken by each step in a chute deployment. Step 2 took 321 seconds, which is more than 50% of the total time. If an application is sensitive to deploying time, we can pre-build and store the image for that application in the private repository, then step 2,3,4 and a large part of step 5 can be eliminated. So that we only need to download the image and start the container to deploy a chute.

all the files for the chute from a file server. The chute deployment takes very long time. Fig. 5.11 illustrates the time taken by each step. Fortunately, the *ParaDrop* gateway can cache the images so that we do not need to download and configure an image to deploy a chute if the chute can share the base image with another deployed chute.

Effectiveness of Resource Management

To evaluate the effectiveness of the resource management of the *ParaDrop* platform, we developed two test chutes and deployed them on a *ParaDrop* AP. Fig. 5.12 shows

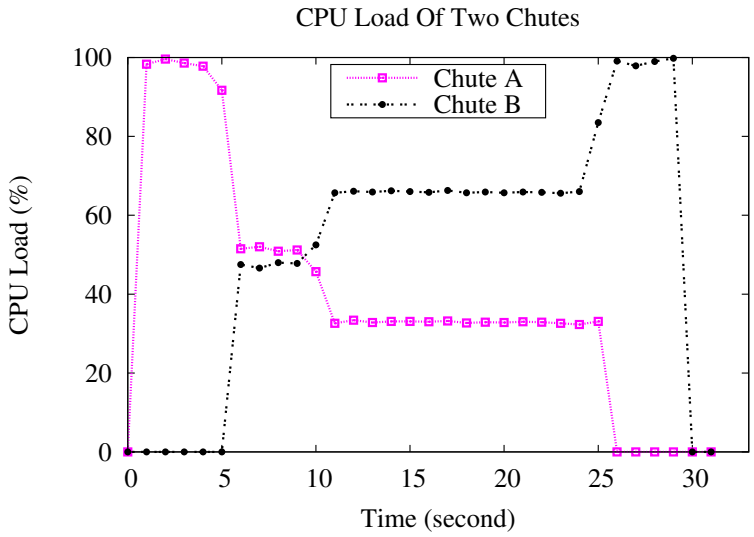


Figure 5.12: The CPU resource management test. We started two chutes from the beginning. Chute A and B have *share* values 512 and 1024 respectively. Both of them can stress test the CPU when they work. Chute A started working from the beginning and became idle after 25 seconds. Chute B started working 5 seconds later, it also kept working for 25 seconds then became idle. In the beginning, chute A's CPU load is about 99%. After 5 seconds, Chute A and B started to compete for CPU resource. They achieved a balance in 11 seconds and Chute B's CPU load was about twice of Chute A's CPU load since then. After chute A became idle, chute B used all the CPU resource and the CPU load was about 99%. Total CPU load of the containers is always about 99% because there was no other computationally intensive task running on the gateway.

the result. Docker uses *cgroups* to group processes running in a container and allows us to manage the resources for containers. When we build a container, we can specify the CPU *share*. The default value is 1024. This value does not mean anything when we talk about it alone. But when two containers both want to use 100% CPU, the *share* values will decide how much share of the CPU that the containers can use. For example, if container A's *share* is 512, container B's *share* is 1024. When two containers are busy running in the same time, container B will have two times share than container A. However, Docker does not limit a container to use free resources. For example, if container B is idle, then container A can use all

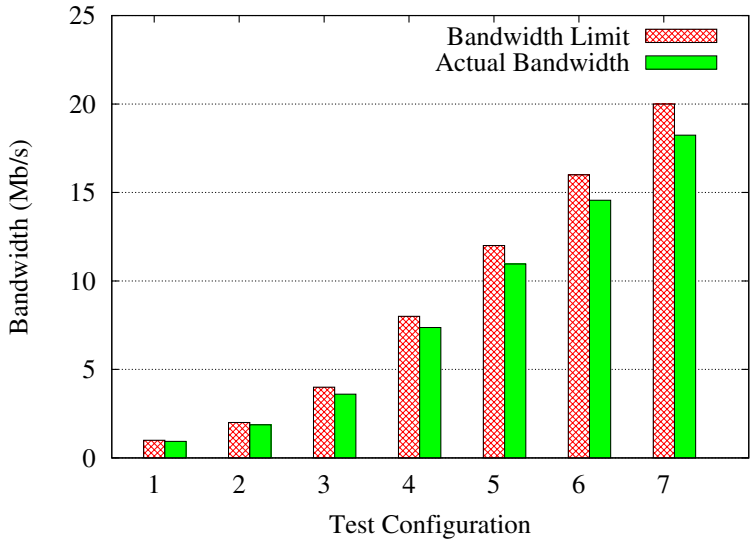


Figure 5.13: Network speed limit test. We launched a test chute including an HTTP server and a 100MB file. In order to avoid the network bandwidth variation of wireless interfaces, we conducted the test with an Ethernet interface. We limited the network bandwidth of the chute to seven different values and tested the actual bandwidth when a user download the large file from the chute. Without the speed limit, the network bandwidth is 89.6Mb/s. We can see the actual bandwidths are not higher than the limits and the discrepancy is small.

available CPU resources. The test results confirm that the container's CPU resource can be effectively managed. Currently, we can specify the CPU *share* value for the chute in the configuration file. It is also possible to change the CPU *share* of the container on-the-fly, though we do not provide this API for now.

By default, every chute can use all of the memory resources available in the gateway. That is not encouraged because it can lead to issues that one chute can easily make the system unstable by allocating too much memory. We can specify the maximum memory that a chute can use in the configuration file. One thing we should note is we can also control the swap available for a chute. By default we disable the swap usage for the chute, so we need to be careful to specify enough (but not too much) memory to a chute. We verified that with a test chute which allocates 256MB memory. When we limited the memory allocated to the chute is

only 200MB, the chute could not start successfully. It would start successfully when the limit is 260MB though.

The network is an important resource for the services deployed on a gateway. All the running chutes share the network bandwidth (LAN and WAN side). Docker (the version we are using) does not provide the support to throttle bandwidth. So we employed the traffic shaping feature of the *tc* command to implement the network resource management for chutes. For the chutes that need network interfaces, we use the *tc* command to limit the bandwidth of these network interfaces for these chutes. Test results in Fig. 5.13 indicate our approach is effective.

A *ParaDrop* gateway has an SD card with 16GB capacity in a typical configuration. By default every container in the Docker can get 10GB of space, that is too much for a chute to run on a *ParaDrop* gateway. We changed that to be 1GB for every chute, and that should be enough for most applications. In the future, we plan to support different quotas for different chutes.

The read/write performance of the SD card is not very high. We need to control the speed that a chute access the file system or else one chute would slow down all other chutes easily. That remains a future work.

Flexibility and Convenience to Deploy Services at the Edge

ParaDrop is highly flexible in deploying edge computing applications. This flexibility is guaranteed by the following design choices:

- The underlying operating system on the *ParaDrop* AP is Snappy Ubuntu. It provides secure and transactional update capabilities to reliably manage the software on the gateways. Furthermore, cloud computing platforms operating systems are quite similar to Snappy Ubuntu. Developers familiar with these systems can easily port over or develop on the *ParaDrop* platform.

- We implemented a virtualized environment based on Docker for the gateway. Developers can select the programming languages, libraries, and platforms based on their experience and requirements. For example, a developer can

use different versions of Python to develop the services. That flexibility makes the *ParaDrop* platform easily acceptable to a vast number of developers. In addition, Docker is a popular virtualization solution for cloud computing service deployment. Therefore, the barrier to port a whole service or some microservices of a large service from a cloud computing platform to the *ParaDrop* platform is very low. The only requirement for services on the *ParaDrop* platform is that they should be able to run on a relatively new version of the Linux kernel. We believe that will not be a problem in practice. Compared to developing an application for Android or iOS, developers utilizing the *ParaDrop* platform do not need to install SDKs or learn new application frameworks.

- The WAMP-based messaging simplified the deployment of the *ParaDrop* gateways. *ParaDrop* leverages both the remote procedure call and publish/subscribe messaging schemes to implement the communications between the *ParaDrop* backend server and *ParaDrop* gateways. Developers do not need to maintain direct connections to the gateways to debug their chutes in deployment.

Scalability

By splitting and distributing the services on the cloud to the *ParaDrop* gateways, *ParaDrop* provides a strong platform to deploy services at large scales. We only transmit latency-sensitive messages with WAMP and transmit other messages with HTTP. We do this so that the WAMP router (crossbar.io) is not the bottleneck of the system even with a large number of APs. The web server in the *ParaDrop* backend can be replicated if necessary to support large-scale deployment.

5.7 Conclusion

ParaDrop is a flexible edge computing platform that users can deploy diverse applications and services at the *extreme* edge of the network. In this chapter, we introduce the three key components of the platform: a flexible hosting substrate in

the Wi-Fi APs that supports multi-tenancy, a cloud-based backend through which such computations are orchestrated across many *ParaDrop* APs, and an API through which third party developers can deploy and manage their computing functions across such different *ParaDrop* APs. We built the system with low-overhead virtualization technology to efficiently use the hardware resources of the *ParaDrop* APs. We implemented effective resource management policies to provide a controlled environment for services running on the *ParaDrop* APs.

We have already conducted tutorials and workshops with *ParaDrop* in multiple forums (at the US Ignite 2014 conference, a GENI Engineering Conference also in 2014, and MobiCom 2017 and 2018) with great success. Users were able to build services such as *SecCam* from scratch, within a few hours, providing some preliminary evidence of its ease of use.

6 Related Work

In this chapter, we discuss various efforts and projects related to the work presented in this thesis. First, we discuss the related work on video transmission optimization and intelligent video analytics. Then we introduce the applications built with edge computing paradigm. Finally, we discuss the work on edge computing infrastructures both in academia and industry.

6.1 Video Transmission Optimizations

Video transmission optimization for IP networks is a classic problem that has attracted research efforts for decades. Researchers have tried various approaches on video coding/transcoding, transmission, and combinations of them.

Advanced Video Coding. Some video encoder implementations support coarse-grained content-aware encoding, e.g., x264 [149] supports tunings optimized for film, animations, etc. The content-aware encoding approach of *DeepRTC* provides fine-grained control over the bit allocation by distinguishing important and unimportant regions of video frames. DNN-based video compression is a new direction that researchers are exploring[64], and it has great potential to provide content-aware optimization. However, speed remains a bottleneck for its usage in RTC applications for now.

Video Transcoding. Video transcoding is critical for video streaming services. Different approaches and architectures have been proposed to implement it for various use cases. Vetro et al. discussed the transcoding of block-based video coding schemes that use hybrid discrete cosine transform (DCT) and motion compensation (MC) [150]. Xin et al. discussed several techniques for reducing the complexity and improving video quality by exploiting the information extracted from the input video bitstream [151]. Unlike their research, we believe that the cascaded decoder and encoder approach is much more straightforward and flexible. As the hardware video encoder improving quality and efficiency and reducing the cost, a cascaded pixel-domain approach is more suitable for practical deployments. For a particular scenario, Youn et al. showed that for point-to-multipoint transcoding, a cascaded video transcoder is more efficient since some parts of the transcoder can be shared [152].

Li et al. introduced a system using cloud transcoding to optimize video streaming service for mobile devices [153]. Video transcoding on a cloud platform is a good solution to transcode a large volume of video data because of its scalability and high throughput. For instance, Amazon, Microsoft, and Telestream Cloud provide cloud transcoding service for users [154, 155, 156]. Netflix also deployed its video transcoding platform on Amazon's cloud [22]. Fouladi et al. built ExCamera, a cloud-based video-processing framework using a functional-programming style [157].

Specialized Hardware for Video Compression. The efficiency issue of general-purpose processors on multimedia applications, including video decoding and encoding, has attracted a lot of research efforts. Various approaches have been studied to improve the hardware efficiency, including specialized instructions [158, 159], specialized architectures [160, 161, 162], GPU offloading [163, 164], application-specific integrated circuit(ASIC) [165], and FPGA-based accelerators [166].

Adaptive Bitrate Video Streaming over HTTP. Adaptive bitrate video streaming is a widely used technique by video streaming service providers to provide high-quality video streaming services. Designing a robust and reliable algorithm to switch bitrate is challenging. Many researchers have proposed adaptation algo-

rithms to achieve a better video quality in dynamic network environments [167, 168, 169]. These works focus on the client side implementation, whereas our work on *VideoCoreCluster* concentrates on the server side. Researchers have also explored the idea of using DNNs to enhance the quality of on-demand streaming services[98].

Real-time Video Transmission. Real-time video applications can adapt the video encoder to generate video streams according to the performance of transmission channels. The sender can obtain the real-time transmission channel information either by monitoring the underlying network performance, or by inferring the performance based on feedback from the receiver. Salsify is a recent research effort to low latency video transmission by tightly integrating video codec with transport protocol [170]. Unlike these systems, *DeepRTC* tries to optimize the end-to-end video quality at both sender and receiver sides. It optimizes the overall system performance by optimizing both the coding and the way information is transmitted.

Cross-layer Optimizations for Video Transmission. Researchers have explored cross-layer optimizations to improve the performance of real-time video transmissions, such as Apex [171] and FlexCast [172]. These solutions optimize video transmission using knowledge of wireless channel conditions. *DeepRTC* can be used in conjunction with them to further boost system performance.

6.2 Edge Computing Architecture and Applications

Many researchers have explored the advantages of edge computing and proposed different approaches to using them. Balan et al. proposed cyber foraging: a mechanism to augment the computational and storage capabilities of mobile devices. Cyber foraging uses opportunistically discovered servers to improve the performance of interactive applications and distributed file system on mobile clients [1]. Satyanarayanan et al. proposed cloudlet, a trusted, resource-rich computer or cluster of computers that's well-connected to the Internet and available for use by nearby mobile devices [2]. Cloudlet can achieve interactive response because of the cloudlet's physical proximity and one-hop network latency. Bonomi et al. discussed Fog Computing, which brings data processing, networking, storage and analytics

closer to mobile devices. They argued that the characteristics of Fog Computing, e.g., low latency and location awareness, very large number of nodes, make it the appropriate platform for many critical IoT services and applications [3].

Some applications were built to leverage the advantages of edge computing architecture. MOCHA is a mobile-cloudlet-cloud architecture that partitions tasks from mobile devices to the cloud and distribute compute load among cloud servers (cloudlet) to minimize the response time [173]. Ha et al. describe the architecture and prototype implementation of an assistive system based on Google Glass devices [174]. They use very powerful machines to build the cloudlet. However, *ParaDrop* uses the hardware platform with medium resources because it has different goals. The targeting applications of *ParaDrop* are those requiring low latency and resource always on and available.

Using edge computing for live video analytics is a hot research topic. Ananthanarayanan et al. discuss the performance requirements of a wide range of video analytics applications, such as traffic, self-driving and smart cars, personal digital assistants, surveillance, and security. They conclude that a geographically distributed architecture of public clouds and edges is the only feasible approach to meeting the strict real-time requirements of large-scale live video analytics [175]. Zhang et al. introduce VideoStorm, a video analytics system that processes thousands of video analytics queries on live video streams over large clusters [176]. Zhang et al. built Vigil, a real-time distributed wireless surveillance system that leverage edge computing to support real-time tracking and surveillance in different scenarios [177]. Wang et al. present OpenFace, an open-source face recognizer whose accuracy approaches that of the best available proprietary recognizers. They built RTFace, a mechanism for denaturing video streams that selectively blurs faces according to specified policies at full frame rates to enable privacy management for live video analytics [178]. Not targeting any specific application, *EdgeEye* is a generic edge computing framework for real-time video analytics. Its goals are usability, flexibility, and high efficiency.

Many research efforts on DNNs primarily focus on improving the training speed and inference accuracy. While application developers care more about inference

speed and easy to use interfaces. Zhao et al. propose a system called Zoo [179] to migrate this gap. They build Zoo based on a numerical computing system written in OCaml language. *EdgeEye* shares a similar goal with Zoo, but we build the system based on available work from industry and open source community.

7 Summary and Future Work

7.1 Summary

With the goals to understand the benefit of edge computing paradigm and the complexity of building an infrastructure for edge computing, we conducted research on edge computing applications, frameworks, and infrastructures.

Our first three pieces of work focus on video related applications demanding high bandwidth and low latency — live video streaming, real-time video communication, and real-time intelligent video analytics. We also apply edge computing on IoT applications with requirements on reliability and high privacy. The evaluation results demonstrate the usefulness of edge computing for those applications.

Our work on *ParaDrop*, which is an open source edge computing platform for research and education purposes, provides us with a valuable opportunity to explore and learn the process to build such a practical system. Based on our experience building *ParaDrop*, the complexities to manage distributed and heterogeneous resources and services are the biggest challenges.

7.2 Discussion

We believe the adoption of edge computing relies on the availability of the tools, frameworks, and platforms. In retrospect, cloud computing was not widely adopted in the industry before cloud providers, such as Amazon EC2, Microsoft Azure, and Google Cloud, give developers access to powerful tools and frameworks to manage resources and services in the cloud platforms. We think the same thing is happening on edge computing. Even though edge computing has a great potential to benefit a lot of applications, its adoption depends on the availability of easy-to-use tools, frameworks, and platforms. Actually, all these cloud providers are also enhancing their tools and frameworks for edge computing applications, especially IoT applications. Examples include AWS IoT Greengrass, Azure IoT Edge, and Google Cloud IoT Edge.

Adopting edge computing does not mean cloud computing will be obsolete. Instead, we believe there will always be applications suited for cloud computing. Some advantages of cloud computing, such as resource elasticity and storage permanence, are crucial for many applications. Edge computing will be a good complement to cloud computing. With the availability of infrastructure for edge computing in the future, we believe many applications will have components in both cloud and edge. Actually, in the process to build and improve the *ParaDrop* platform, we emphasize the ability to support cloud-edge hybrid applications, so that developers can combine the strengths of both cloud and edge computing to build useful applications.

7.3 Future Work

Edge computing is an evolving field with many potential applications in different areas. We have only explored a very small portion of them in our work. As more smart devices are being deployed in the "edge", edge computing is becoming more important for many services and applications. In our current research on the edge computing infrastructure, we reuse the technologies designed for cloud computing.

However, because of the unique requirements and challenges in edge computing, more in-depth and fundamental research on the edge computing infrastructure is needed. In this section, we envision the future work can be done on edge computing regarding application and infrastructure.

More Edge Computing Applications

Our work on *EdgeEye* shows some potential of edge computing on AR applications, and we plan to continue exploring the applications of edge computing on AR systems. With the low-latency offloading support, we can build more complex mobile AR applications which are inefficient or even infeasible for existing mobile devices.

Instead of simply offloading or deploying tasks and service into the edge, we can split tasks with more fine-grained granularity and distribute smaller subtasks into the cloud, edge, and end devices to maximize the overall system performance and efficiency. For example, some researchers have explored the idea to split a deep neural network to multiple pieces, where each piece has some layers of the original neural network. Then the pieces are deployed into the cloud, edge, and end devices to complete a task. How to best split the original task and distribute the subtasks are interesting problems to work on.

The work covered in this thesis does not use contextual information from the edge network, which is an advantage of edge computing. In future work, we will try to leverage such information, e.g., wireless channel information, to optimize application performance.

Edge Computing Infrastructures

Polishing *ParaDrop*. *ParaDrop* is still an ongoing project, we are in the progress to make it easy to be used by developers working on different domains. Till recently, we have been focusing on its overall functionalities as an edge computing platform. We plan to polish it to further improve its capabilities, e.g., we can integrate specialized hardware accelerators for DNNs into the current prototype hardware.

Accelerating DNN inference is an important usage case for edge computing. Infrastructure providers often use GPUs to accelerate DNN's in the cloud, but they may not be suitable for application requiring low latency. GPUs can achieve high throughput by batching evaluations and that is good for offline training. However, for the online inference, which is common in edge computing settings, batching is hard to implement because requests often arrive one at a time. To overcome that challenge, Microsoft chose FPGAs to build BrainWave for real-time AI in the cloud which can achieve both high throughput and low latency [180]. Similar work needs to done for edge computing as well. Specialized hardware support for real-time AI in the *ParaDrop* platform will give developers more power to build edge services.

Another feature we can add to the *ParaDrop* platform is the Function as a Service (FaaS) support. That support can simplify the developer's job to build services for the *ParaDrop* platform.

Other forms of edge computing. Edge computing has a broad and blurry definition. In the *ParaDrop* project, we explored a specific point in the network edge — WiFi APs. Researchers are working on other forms of edge computing. For example, the industry is interested in telecom-centric edge computing — Multi-access Edge Computing (MEC, also known as Mobile Edge Computing) [4]. MEC proposes to provide computational and storage resource in the access network, and it is recognized by the European 5G PPP (5G Infrastructure Public Private Partnership) research body as one of the key emerging technologies for 5G networks [5]. It is an interesting field to explore and the research will be impactful [181]. Compared with the current 4G LTE technology, 5G aims to provide orders of magnitude improvement on latency and bandwidth. Radio access network latency will be only 2~5 milliseconds, and peak data rates can be up to 10 Gbps [182]. With the resources deployed in the access network, such a low latency and high bandwidth will make many latency sensitive applications possible for mobile devices — such as virtual reality and augmented reality. 5G networks will also support a higher number of connecting devices than 4G networks, and edge computing can help to process the data from those devices locally and thus reduce the possibility to create congestions in the core networks. Operators will have strong economic motivations to host

resources in the access network and give developers access to those resources to build applications. They will be edge computing Infrastructure as a Service (IaaS) providers instead of just providing data pipe to users and cloud computing service providers.

Underlying technologies for edge computing. Docker and Linux containers are widely used for cloud computing. We leverage them to build the *ParaDrop* platform, but we did not research on the fundamental requirements of edge services on isolation and migration. We plan to work on that in the future work, especially to understand the unique problems of supporting lightweight multi-tenancy for edge computing, for example, hardware diversity would be a big problem needs to be solved.

Security of edge computing infrastructure is also an important future work. Unlike cloud computing, the resources of an edge computing platform might not in the premise of the platform owner, and developers might have physical access to the resources. More research is needed to figure out how to restrict access to sensitive information and guarantee correctness and reliability.

Bibliography

[1] Rajesh Balan, Jason Flinn, Mahadev Satyanarayanan, Shafeeq Sinnamohideen, and Hen-I Yang. The case for cyber foraging. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 87–92. ACM, 2002.

[2] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *Pervasive Computing, IEEE*, 8(4):14–23, 2009.

[3] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.

[4] Tarik Taleb, Konstantinos Samdanis, Badr Mada, Hannu Flinck, Sunny Dutta, and Dario Sabella. On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration. *IEEE Communications Surveys & Tutorials*, 19(3):1657–1681, 2017.

[5] Yun Chao Hu, Milan Patel, Dario Sabella, Nurit Sprecher, and Valerie Young. Mobile edge computing—a key technology towards 5g. *ETSI White Paper*, 11, 2015.

[6] Peng Liu, Jongwon Yoon, Lance Johnson, and Suman Banerjee. Greening the video transcoding service with low-cost hardware transcoders. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, pages 407–419, 2016.

[7] Peng Liu, Bozhao Qi, and Suman Banerjee. EdgeEye: An edge service framework for real-time intelligent video analytics. In *Proceedings of the 1st International Workshop on Edge Systems, Analytics and Networking*, pages 1–6. ACM, 2018.

[8] Peng Liu, Dale Willis, and Suman Banerjee. ParaDrop: Enabling lightweight multi-tenancy at the network's extreme edge. In *Edge Computing (SEC), IEEE/ACM Symposium on*, pages 1–13. IEEE, 2016.

[9] Peng Liu, Lance Hartung, and Suman Banerjee. Lightweight multitenancy at the network's extreme edge. *Computer*, 50(10):50–57, 2017.

[10] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill 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.

[11] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[12] Steffen Gebert, Rastin Pries, Daniel Schlosser, and Klaus Heck. Internet Access Traffic Measurement and Analysis. In *Proc. of ACM TMA*, 2012.

[13] Janko Roettgers. To stream everywhere, Netflix encodes each movie 120 times, 2012. `https://gigaom.com/2012/12/18/netflix-encoding/`.

[14] Andrew Banks and Rahul Gupta. Mqtt version 3.1. 1. *OASIS Standard*, 2014.

[15] Thomas Stockhammer. Dynamic adaptive streaming over http–: standards and design principles. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 133–144. ACM, 2011.

[16] Apple Inc. HTTP Live Streaming, 2019. `https://developer.apple.com/streaming/`.

[17] Ishfaq Ahmad, Xiaohui Wei, Yu Sun, and Ya-Qin Zhang. Video transcoding: an overview of various techniques and research issues. *Multimedia, IEEE Transactions on*, 7(5):793–804, 2005.

[18] ITUT Draft. recommendation and final draft international standard of joint video specification (itu-t rec. h. 264| iso/iec 14496-10 avc). *Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, JVTG050*, 33, 2003.

[19] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 37–47. ACM, 2010.

[20] Qualcomm. Enabling the Full 4K Mobile Experience: System Leadership, 2019. `https://www.qualcomm.com/documents/enabling-full-4k-mobile-experience-system-leadership`.

[21] Alberto Duenas. State of the art of video on smartphone, 2014. `https://ngcodec.com/news/2014-3-13-state-of-the-art-of-video-on-smartphone`.

[22] Kevin McEntee. Netflix's encoding transformation, 2019. `http://www.slideshare.net/AmazonWebServices/med202-netflixtranscodingtransformation`.

[23] H.264/AVC Reference Software, 2019. `http://iphome.hhi.de/suehring/`.

[24] VideoLAN Organization. x264, 2019. `http://www.videolan.org/developers/x264.html`.

[25] Loren Merritt and Rahul Vanam. Improved rate control and motion estimation for h.264 encoder. In *Image Processing, 2007. ICIP 2007. IEEE International Conference on*, volume 5, pages V–309. IEEE, 2007.

[26] Cisco. Openh264, 2019. `http://www.openh264.org/`.

[27] Mainconcept, 2019. `https://www.mainconcept.com/`.

[28] Intel. Intel integrated performance primitives (intel ipp) & their applications, 2019. https://software.intel.com/en-us/intel-ipp.

[29] Nvidia. Nvidia video codec sdk, 2019. https://developer.nvidia.com/nvidia-video-codec-sdk.

[30] AMD. Introducing the video coding engine (vce), 2014. http://developer.amd.com/community/blog/2014/02/19/introducing-video-coding-engine-vce/.

[31] Xilinx. H.246 4k video encoder module, 2019. https://www.xilinx.com/products/boards-and-kits/1-fod2l8.html.

[32] MSU Graphics & Media Lab (Video Group). Eighth mpeg-4 avc/h.264 video codecs comparison - standard version, 2019. http://www.compression.ru/video/codec_comparison/h264_2012/.

[33] Jan Newmarch. Programming audiovideo on the raspberry pi gpu, 2016. https://jan.newmarch.name/RPi/.

[34] Raspberry pi firmware, 2019. https://github.com/raspberrypi/firmware.

[35] Source code for arm side libraries for interfacing to raspberry pi gpu, 2019. https://github.com/raspberrypi/userland.

[36] Google. Protocol buffers, 2019. https://developers.google.com/protocol-buffers/?hl=en.

[37] Adobe. Real-time messaging protocol (rtmp) specification, 2019. http://www.adobe.com/devnet/rtmp.html.

[38] The Khronos Group Inc. Openmax integration layer application programming interface specification, 2008. https://www.khronos.org/registry/omxil/specs/OpenMAX_IL_1_1_2_Specification.pdf.

[39] GStreamer. Gstreamer: open source multimedia framework, 2018. https://gstreamer.freedesktop.org/.

[40] gst-omx, 2019. https://github.com/pliu6/gst-omx.

[41] The Apache Software Foundation. Apache http server project, 2019. https://httpd.apache.org/.

[42] nginx, 2019. https://www.nginx.com/.

[43] Node.js Foundation. Node.js, 2019. https://nodejs.org/en/.

[44] Mqtt.js - node and javascript mqtt client and parser, 2019. https://github.com/mqttjs.

[45] Mqtt c++ client for posix and windows, 2019. https://eclipse.org/paho/clients/cpp/.

[46] Eclipse mosquitto - an open source mqtt v3.1/v3.1.1 broker, 2019. http://mosquitto.org/.

[47] Nginx-based media streaming server, 2015. https://github.com/pliu6/nginx-rtmp-module.

[48] Buildroot - making embedded linux easy, 2019. https://buildroot.org/.

[49] Methodology for the subjective assessment of the quality of television pictures, 2012. https://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.500-13-201201-I!!PDF-E.pdf.

[50] ITU-T. P.910 : Subjective video quality assessment methods for multimedia applications, 2008. https://www.itu.int/rec/T-REC-P.910/en.

[51] Quan Huynh-Thu and Mohammed Ghanbari. Scope of validity of psnr in image/video quality assessment. *Electronics letters*, 44(13):800–801, 2008.

[52] Zhou Wang, Alan Conrad Bovik, Hamid Rahim Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *Image Processing, IEEE Transactions on*, 13(4):600–612, 2004.

[53] Yuv video sequences, 2019. http://trace.eas.asu.edu/yuv/.

[54] Intel. Intel core i5-4570 processor, 2019. http://ark.intel.com/products/75043/Intel-Core-i5-4570-Processor-6M-Cache-up-to-3_60-GHz.

[55] Tony DiCola. Embedded linux board comparison, 2019. https://learn.adafruit.com/downloads/pdf/embedded-linux-board-comparison.pdf.

[56] VNI Cisco. Cisco visual networking index: Forecast and trends, 2017–2022. *White Paper*, 2018.

[57] Barry G Haskell, Atul Puri, and Arun N Netravali. *Digital video: an introduction to MPEG-2*. Springer Science & Business Media, 1996.

[58] Thomas Wiegand, Gary J Sullivan, Gisle Bjontegaard, and Ajay Luthra. Overview of the h. 264/avc video coding standard. *IEEE Transactions on circuits and systems for video technology*, 13(7):560–576, 2003.

[59] Gary J Sullivan, Jens-Rainer Ohm, Woo-Jin Han, Thomas Wiegand, et al. Overview of the high efficiency video coding(hevc) standard. *IEEE Transactions on circuits and systems for video technology*, 22(12):1649–1668, 2012.

[60] Jim Bankoski, Paul Wilkins, and Yaowu Xu. Technical overview of vp8, an open source video codec for the web. In *2011 IEEE International Conference on Multimedia and Expo*, pages 1–6. IEEE, 2011.

[61] Debargha Mukherjee, Jim Bankoski, Adrian Grange, Jingning Han, John Koleszar, Paul Wilkins, Yaowu Xu, and Ronald Bultje. The latest open-source video codec vp9-an overview and preliminary results. In *Picture Coding Symposium (PCS), 2013*, pages 390–393. IEEE, 2013.

[62] Alliance for Open Media. *Get started creating products with AV1*, 2018. https://aomedia.org/av1-features/get-started/.

[63] Chao-Yuan Wu, Nayan Singhal, and Philipp Krahenbuhl. Video compression through image interpolation. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 416–431, 2018.

[64] Oren Rippel, Sanjay Nair, Carissa Lew, Steve Branson, Alexander G Anderson, and Lubomir Bourdev. Learned video compression. *arXiv preprint arXiv:1811.06981*, 2018.

[65] Andrey Ignatov, Radu Timofte, William Chou, Ke Wang, Max Wu, Tim Hartley, and Luc Van Gool. Ai benchmark: Running deep neural networks on android smartphones. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 0–0, 2018.

[66] Iain E Richardson. *The H.264 advanced video compression standard*. John Wiley & Sons, 2011.

[67] Xiph.org. *Xiph.org Video Test Media [derf's collection*, 2018. `https://media.xiph.org/video/derf/`.

[68] WebRTC. *WebRTC*, 2018. `https://webrtc.org/`.

[69] Yang Liu, Zheng Guo Li, and Yeng Chai Soh. Region-of-interest based resource allocation for conversational video communication of h.264/avc. *IEEE transactions on circuits and systems for video technology*, 18(1):134–139, 2008.

[70] Laurent Itti, Christof Koch, and Ernst Niebur. A model of saliency-based visual attention for rapid scene analysis. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (11):1254–1259, 1998.

[71] Tie Liu, Zejian Yuan, Jian Sun, Jingdong Wang, Nanning Zheng, Xiaoou Tang, and Heung-Yeung Shum. Learning to detect a salient object. *IEEE Transactions on Pattern analysis and machine intelligence*, 33(2):353–367, 2011.

[72] Wenguan Wang, Jianbing Shen, and Ling Shao. Video salient object detection via fully convolutional networks. *IEEE Transactions on Image Processing*, 27(1):38–49, 2018.

[73] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[74] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. Ssd: Single shot multibox detector. In *European conference on computer vision*, pages 21–37. Springer, 2016.

[75] Stefanos Zafeiriou, Cha Zhang, and Zhengyou Zhang. A survey on face detection in the wild: past, present and future. *Computer Vision and Image Understanding*, 138:1–24, 2015.

[76] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–I. IEEE, 2001.

[77] OpenCV. *Face Detection using Haar Cascades*, 2018. `https://docs.opencv.org/3.4.2/d7/d8b/tutorial_py_face_detection.html`.

[78] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.

[79] Apple Inc. *An On-device Deep Neural Network for Face Detection*, 2018. `https://machinelearning.apple.com/2017/11/16/face-detection.html`.

[80] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[81] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *arXiv preprint arXiv:1511.00561*, 2015.

[82] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *arXiv preprint arXiv:1606.00915*, 2016.

[83] Alberto Garcia-Garcia, Sergio Orts-Escolano, Sergiu Oprea, Victor Villena-Martinez, and Jose Garcia-Rodriguez. A review on deep learning techniques applied to semantic segmentation. *arXiv preprint arXiv:1704.06857*, 2017.

[84] FFmpeg. *FFmpeg Scaler Documentation*, 2019. `https://ffmpeg.org/ffmpeg-scaler.html`.

[85] Chao Dong, Chen Change Loy, Kaiming He, and Xiaoou Tang. Learning a deep convolutional network for image super-resolution. In *European conference on computer vision*, pages 184–199. Springer, 2014.

[86] Chao Dong, Chen Change Loy, and Xiaoou Tang. Accelerating the super-resolution convolutional neural network. In *European Conference on Computer Vision*, pages 391–407. Springer, 2016.

[87] Wenzhe Shi, Jose Caballero, Ferenc Huszár, Johannes Totz, Andrew P Aitken, Rob Bishop, Daniel Rueckert, and Zehan Wang. Real-time single image and video super-resolution using an efficient sub-pixel convolutional neural network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1874–1883, 2016.

[88] Jiwon Kim, Jung Kwon Lee, and Kyoung Mu Lee. Accurate image super-resolution using very deep convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1646–1654, 2016.

[89] Jiwon Kim, Jung Kwon Lee, and Kyoung Mu Lee. Deeply-recursive convolutional network for image super-resolution. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1637–1645, 2016.

[90] Ying Tai, Jian Yang, and Xiaoming Liu. Image super-resolution via deep recursive residual network. In *Proceedings of the IEEE Conference on Computer vision and Pattern Recognition*, pages 3147–3155, 2017.

[91] Wei-Sheng Lai, Jia-Bin Huang, Narendra Ahuja, and Ming-Hsuan Yang. Deep laplacian pyramid networks for fast and accurate super-resolution. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 624–632, 2017.

[92] Tong Tong, Gen Li, Xiejie Liu, and Qinquan Gao. Image super-resolution using dense skip connections. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4799–4807, 2017.

[93] Christian Ledig, Lucas Theis, Ferenc Huszár, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, et al. Photo-realistic single image super-resolution using a generative adversarial network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4681–4690, 2017.

[94] Bee Lim, Sanghyun Son, Heewon Kim, Seungjun Nah, and Kyoung Mu Lee. Enhanced deep residual networks for single image super-resolution. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 136–144, 2017.

[95] Namhyuk Ahn, Byungkon Kang, and Kyung-Ah Sohn. Fast, accurate, and lightweight super-resolution with cascading residual network. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 252–268, 2018.

[96] Armin Kappeler, Seunghwan Yoo, Qiqin Dai, and Aggelos K Katsaggelos. Video super-resolution with convolutional neural networks. *IEEE Transactions on Computational Imaging*, 2(2):109–122, 2016.

[97] Jose Caballero, Christian Ledig, Andrew Aitken, Alejandro Acosta, Johannes Totz, Zehan Wang, and Wenzhe Shi. Real-time video super-resolution with spatio-temporal networks and motion compensation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4778–4787, 2017.

[98] Hyunho Yeo, Youngmok Jung, Jaehong Kim, Jinwoo Shin, and Dongsu Han. Neural adaptive content-aware internet video delivery. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 645–661, 2018.

[99] The WebM Project. *WebM: an open web media project*, 2018. `https://www.webmproject.org/`.

[100] *Tensorflow Face Detector*, 2019. `https://github.com/yeephycho/tensorflow-face-detection`.

[101] TensorFlow. *DeepLab: Deep Labelling for Semantic Image Segmentation*, 2018. `https://github.com/tensorflow/models/tree/master/research/deeplab`.

[102] Marc Leeman. *Experiences with gstreamer/webrtc*, 2018. `https://gstconf.ubicast.tv/videos/experiences-with-gstreamer-webrtc/`.

[103] Radu Timofte, Eirikur Agustsson, Luc Van Gool, Ming-Hsuan Yang, and Lei Zhang. Ntire 2017 challenge on single image super-resolution: Methods and results. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 114–125, 2017.

[104] Nvidia. *NVIDIA TensorRT, Programmable Inference Accelerator*, 2019. `https://developer.nvidia.com/tensorrt`.

[105] Chao Dong, Yubin Deng, Chen Change Loy, and Xiaoou Tang. Compression artifacts reduction by a deep convolutional network. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 576–584, 2015.

[106] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[107] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, et al. Going deeper with convolutions. Cvpr, 2015.

[108] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[109] resin.io. Resin.io homepage, 2018. `https://resin.io/`.

[110] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[111] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[112] Ronan Collobert, Samy Bengio, and Johnny Mariéthoz. Torch: a modular machine learning software library. Technical report, Idiap, 2002.

[113] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[114] Intel. Intel's deep learning inference engine developer guide, 2017. `https://software.intel.com/en-us/inference-engine-devguide`.

[115] Nvidia. Nvidia digits, interactive deep learning gpu training system, 2018. `https://developer.nvidia.com/digits`.

[116] Nvidia. Detectnet: Deep neural network for object detection in digits, 2016. `https://devblogs.nvidia.com/detectnet-deep-neural-network-object-detection-digits/`.

[117] Samsung. Iot.js - a framework for internet of things, 2018. `http://iotjs.net/`.

[118] Bocoup. Johnny-five: The javascript robotics and iot platform, 2018. `http://johnny-five.io/`.

[119] Luis López Fernández, Miguel París Díaz, Raúl Benítez Mejías, Francisco Javier López, and José Antonio Santos. Kurento: a media server technology for convergent www/mobile real-time multimedia communications supporting webrtc. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2013 IEEE 14th International Symposium and Workshops on a*, pages 1–6. IEEE, 2013.

[120] Adam Bergkvist, Daniel C Burnett, Cullen Jennings, Anant Narayanan, and Bernard Aboba. Webrtc 1.0: Real-time communication between browsers. *Working draft, W3C*, 91, 2012.

[121] BVLC. Caffe: a fast open framework for deep learning, 2018. `https://github.com/BVLC/caffe`.

[122] NVIDIA. Technical overview: Nvidia deep learning platform, 2018. `https://images.nvidia.com/content/pdf/inference-technical-overview.pdf`.

[123] coturn. Free open source implementation of turn and stun server, 2018. `https://github.com/coturn/coturn`.

[124] Shan Tang. A list of ics and ips for ai, machine learning and deep learning, 2018. `https://basicmi.github.io/Deep-Learning-Processor-List/`.

[125] Amazon. The world's first deep learning enabled video camera for developers, 2018. `https://aws.amazon.com/deeplens/`.

[126] Google. Google clips, 2017. https://store.google.com/us/product/google_clips?hl=en-US.

[127] Canonical Ltd. Infrastructure for container projects, 2019. https://linuxcontainers.org/.

[128] Ubuntu. Ubuntu Documentation - LXC, 2019. https://help.ubuntu.com/lts/serverguide/lxc.html.

[129] Docker. Enterprise container platform for high-velocity innovation, 2019. https://www.docker.com.

[130] Crossbar.io Technologies GmbH. Wamp: The Web Application Messaging Protocol, 2019. http://wamp-proto.org/.

[131] PC Engines. PC Engines apu platform, 2019. https://www.pcengines.ch/apu.htm.

[132] Neil Klingensmith, Joseph Bomber, and Suman Banerjee. Hot, cold and in between: enabling fine-grained environmental control in homes for efficiency and comfort. In *Proceedings of the 5th international conference on Future energy systems*, pages 123–132. ACM, 2014.

[133] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux symposium*, volume 1, pages 225–230, 2007.

[134] Kiryong Ha, Padmanabhan Pillai, Wolfgang Richter, Yoshihisa Abe, and Mahadev Satyanarayanan. Just-in-time provisioning for cyber foraging. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 153–166. ACM, 2013.

[135] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and linux containers. In *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium On*, pages 171–172. IEEE, 2015.

[136] Junjiro R. Okajima. Aufs4 – advanced multi layered unification filesystem version 5.x, 2019. http://aufs.sf.net.

[137] Ubuntu. Ubuntu Core, 2019. https://developer.ubuntu.com/core.

[138] Crossbar.io. Autobahn libraries, 2019. https://crossbar.io/autobahn/.

[139] Crossbar.io. Networking for apps, 2019. http://crossbar.io/.

[140] Twisted matrix labs: Building the engine of your Internet, 2019. https://twistedmatrix.com/trac/.

[141] Kristina Chodorow. *MongoDB: the definitive guide*. " O'Reilly Media, Inc.", 2013.

[142] Docker. Use the Device Mapper storage driver, 2019. https://docs.docker.com/storage/storagedriver/device-mapper-driver/.

[143] D-Link Corporation. mydlink, 2019. https://www.mydlink.com/entrance.

[144] YAML: YAML Ain't Markup Language, 2019. https://yaml.org.

[145] Yong Liu, Yang Guo, and Chao Liang. A survey on peer-to-peer video streaming systems. *Peer-to-peer Networking and Applications*, 1(1):18–28, 2008.

[146] Florian Fainelli. The openwrt embedded development framework. In *Proceedings of the Free and Open Source Software Developers European Meeting*, 2008.

[147] Rusty Russel. lguest: Implementing the little linux hypervisor. *OLS*, 7:173–178, 2007.

[148] Dale Willis, Arkodeb Dasgupta, and Suman Banerjee. Paradrop: a multi-tenant platform to dynamically install third party services on wireless gateways. In *Proceedings of the 9th ACM workshop on Mobility in the evolving internet architecture*, pages 43–48. ACM, 2014.

[149] FFmpeg. *H.264 Video Encoding Guide*, 2019. `https://trac.ffmpeg.org/wiki/Encode/H.264`.

[150] Anthony Vetro, Charilaos Christopoulos, and Huifang Sun. Video transcoding architectures and techniques: an overview. *Signal Processing Magazine, IEEE*, 20(2):18–29, 2003.

[151] Jun Xin, Chia-Wen Lin, and Ming-Ting Sun. Digital video transcoding. *Proceedings of the IEEE*, 93(1):84–97, 2005.

[152] Jeongnam Youn, Jun Xin, Ming-Ting Sun, and Ya-Qin Zhang. Video transcoding for multiple clients. In *Visual Communications and Image Processing 2000*, pages 76–85. International Society for Optics and Photonics, 2000.

[153] Zhenhua Li, Yan Huang, Gang Liu, Fuchen Wang, Zhi-Li Zhang, and Yafei Dai. Cloud transcoder: Bridging the format and resolution gap between internet videos and mobile devices. In *Proceedings of the 22nd international workshop on Network and Operating System Support for Digital Audio and Video*, pages 33–38. ACM, 2012.

[154] Microsoft. Azure media services, 2019. `https://azure.microsoft.com/en-us/services/media-services/encoding/`.

[155] Amazon. Amazon elastic transcoder, 2019. `https://aws.amazon.com/elastictranscoder/`.

[156] Telestream. Enabling future-proof media processing, 2019. `https://cloud.telestream.net/`.

[157] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 363–376, 2017.

[158] Alex Peleg and Uri Weiser. Mmx technology extension to the intel architecture. *Micro, IEEE*, 16(4):42–50, 1996.

[159] Marc-André Daigneault, JM Pierre Langlois, and Jean Pierre David. Application specific instruction set processor specialized for block motion estimation. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, pages 266–271. IEEE, 2008.

[160] Sung Dae Kim and Myung H Sunwoo. Asip approach for implementation of h.264/avc. *Journal of Signal Processing Systems*, 50(1):53–67, 2008.

[161] Sangwon Seo, Mark Woh, S Mahlke, T Mudge, Sunfaram Vijay, and Chaitali Chakrabarti. Customizing wide-simd architectures for h.264. In *Systems, Architectures, Modeling, and Simulation, 2009. SAMOS'09. International Symposium on*, pages 172–179. IEEE, 2009.

[162] Mark Woh, Sangwon Seo, Scott Mahlke, Trevor Mudge, Chaitali Chakrabarti, and Krisztian Flautner. Anysp: anytime anywhere anyway signal processing. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 128–139. ACM, 2009.

[163] Yu-Cheng Lin, Pei-Lun Li, Chin-Hsiang Chang, Chi-Ling Wu, You-Ming Tsao, and Shao-Yi Chien. Multi-pass algorithm of motion estimation in video encoding for generic gpu. In *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, pages 4–pp. IEEE, 2006.

[164] Wei-Nien Chen and Hsueh-Ming Hang. H. 264/avc motion estimation implmentation on compute unified device architecture (cuda). In *Multimedia and Expo, 2008 IEEE International Conference on*, pages 697–700. IEEE, 2008.

[165] Yu-Kun Lin, De-Wei Li, Chia-Chun Lin, Tzu-Yun Kuo, Sian-Jin Wu, Wei-Cheng Tai, Wei-Cheng Chang, and Tian-Sheuan Chang. A 242mw, 10mm 2 1080p h.264/avc high profile encoder chip. In *Proceedings of the 45th annual Design Automation Conference*, pages 78–83. ACM, 2008.

[166] Olli Lehtoranta, Erno Salminen, Ari Kulmala, Marko Hännikäinen, and Timo D Hämäläinen. A parallel mpeg-4 encoder for fpga based multiprocessor soc. In *Field Programmable Logic and Applications, 2005. International Conference on*, pages 380–385. IEEE, 2005.

[167] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 97–108. ACM, 2012.

[168] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 187–198. ACM, 2014.

[169] Xiaoqi Yin, Abhishek Jindal, Vyas Sekar, and Bruno Sinopoli. A control-theoretic approach for dynamic adaptive video streaming over http. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 325–338. ACM, 2015.

[170] Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S Wahby, and Keith Winstein. Salsify: Low-latency network video through tighter integration between a video codec and a transport protocol. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. USENIX Association, 2018.

[171] Sayandeep Sen, Syed Gilani, Shreesha Srinath, Stephen Schmitt, and Suman Banerjee. Design and implementation of an approximate communication system for wireless media applications. *ACM SIGCOMM Computer Communication Review*, 41(4):15–26, 2011.

[172] Siripuram Aditya and Sachin Katti. Flexcast: Graceful wireless video streaming. In *Proceedings of the 17th annual international conference on Mobile computing and networking*, pages 277–288. ACM, 2011.

[173] Tolga Soyata, Rajani Muraleedharan, Colin Funai, Minseok Kwon, and Wendi Heinzelman. Cloud-vision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture. In *Computers and Communications (ISCC), 2012 IEEE Symposium on*, pages 000059–000066. IEEE, 2012.

[174] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 68–81. ACM, 2014.

[175] Ganesh Ananthanarayanan, Paramvir Bahl, Peter Bodík, Krishna Chintalapudi, Matthai Philipose, Lenin Ravindranath, and Sudipta Sinha. Real-time video analytics: The killer app for edge computing. *Computer*, 50(10):58–67, 2017.

[176] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J Freedman. Live video analytics at scale with approximation and delay-tolerance. In *NSDI*, volume 9, page 1, 2017.

[177] Tan Zhang, Aakanksha Chowdhery, Paramvir Victor Bahl, Kyle Jamieson, and Suman 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. ACM, 2015.

[178] Junjue Wang, Brandon Amos, Anupam Das, Padmanabhan Pillai, Norman Sadeh, and Mahadev Satyanarayanan. A scalable and privacy-aware iot service for live video analytics. In *Proceedings of the 8th ACM on Multimedia Systems Conference*, pages 38–49. ACM, 2017.

[179] Jianxin Zhao, Richard Mortier, Jon Crowcroft, and Liang Wang. User-centric composable services: A new generation of personal data analytics. *arXiv preprint arXiv:1710.09027*, 2017.

[180] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 1–14. IEEE Press, 2018.

[181] Sami Kekki, Walter Featherstone, Yonggang Fang, Pekka Kuure, Alice Li, Anurag Ranjan, Debashish Purkayastha, Feng Jiangping, Danny Frydman, Gianluca Verin, et al. Mec in 5g networks. *ETSI White Pap*, 28, 2018.

[182] Ekram Hossain and Monowar Hasan. 5g cellular: key enabling technologies and research challenges. *arXiv preprint arXiv:1503.00674*, 2015.