

An Application Platform for Wearable Cognitive Assistance

Zhuo Chen

CMU-CS-18-104

April 2018

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:

Mahadev Satyanarayanan (Chair)

Daniel P. Siewiorek

Martial Hebert

Padmanabhan Pillai (Intel)

*Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.*

Copyright © 2018 Zhuo Chen

This research was sponsored by the National Science Foundation under grant numbers CNS-0833882, CNS-1518865, and IIS-1065336; by the US Department of Defense Advanced Research Projects Agency (DARPA) under grant number FA8721-05-C-0003; and by Intel Corporation, Santa Clara under grants A018540296213611 and ISTC-CC. Additional support was provided by Google, Vodafone, Deutsche Telekom, Verizon, Crown Castle, NTT, and the Conklin Kistler family fund. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Mobile Computing, Wearable Computing, Ubiquitous Computing, Computer Vision, Google Glass, Cloudlet, Cognitive Assistance, Coarse-grain Parallelism, Cyber Foraging, Augmented Reality

To my parents.

Abstract

Wearable cognitive assistance applications can provide guidance for many facets of a user's daily life. This thesis targets the enabling of a new genre of such applications that require both heavy computation and very low response time on inputs from mobile devices. The core contribution of this thesis is the design, implementation, and evaluation of Gabriel, an application platform that simplifies the creation of and experimentation with this new genre of applications. An essential capability of this platform is to use cloudlets for computation offloading to achieve low latency. By implementing several prototype applications on top of Gabriel, the thesis evaluates the system performance of Gabriel under various system conditions. It also shows how Gabriel is capable of exploiting coarse grain parallelism on cloudlets to improve system performance, and conserving energy on mobile devices based on user context.

Acknowledgments

I would like to first thank my adviser, Prof. Mahadev Satyanarayanan (Satya). It has been a great pleasure and honor to work with him for the past six years, and to learn from his experience and insights in being a computer scientist. Satya has also been a great mentor of my life, and has provided invaluable help and guidance in my career decision. I would also like to thank Padmanabhan (Babu) Pillai, who has almost been my co-adviser during my Ph.D. Babu has tremendous knowledge on almost every aspect of computer systems, and I always learn a lot in talking with him. Also special thanks to him for staying with me late at night for paper deadlines. In addition, I would like to thank my thesis committee members, Daniel P. Siewiorek and Martial Hebert. It was a fortune to meet with Dan at weekly Elijah meetings to learn from his years of experience in building wearable systems. It was also fortunate to be guided by Martial's expertise on the computer vision aspect of my thesis.

My Ph.D journey has also been surrounded by many talented and diligent people in Satya's research group. I have been constantly inspired by their ideas, motivated by their passion, and sharpened by their critical thinking. I would like to especially thank Yoshihisa Abe, Brandon Amos, Tom Eiszler, Ziqiang Feng, Shilpa George, Benjamin Gilbert, Jan Harkes, Wenlu Hu, Wolfgang Richter, and Junjue Wang. Last but not least, I would want to thank Chase Klingensmith for his always prompt assistance for group administrative matters.

Throughout my Ph.D., I have been lucky to collaborate with people outside of group or outside of CMU. They have brought in different perspectives into my research, and have helped me in different aspects of my thesis. Thank you, Yuvraj Agarwal, Mihir Bala, Da-Yoon Chung, Nigel Davies, Khalid Elgazzar, Junchen Jiang, Guenter Klas, Roberta Klatzky, Grace Lewis, Tan Li, Jeffrey Liu, Ishan Misra, Grace Lewis, Rolf Schuster, Srinivasan Seshan, Pieter Simoens, Brandon Tyler, Guanhang Wu, Yu Xiao, Ben Zhang, Huanchen Zhang, Siyan Zhao.

Finally, I would like to thank my parents for being supportive along the way. They have brought their understanding, sincere suggestion, and consideration in the ups and downs of this long journey. I could not have finished the Ph.D. without them, and am thankful to have them as my parents.

Contents

Acronyms	xix
1 Introduction	1
1.1 Thesis Statement	3
1.2 Technology Advances for Wearable Cognitive Assistance	4
1.3 The Augmented Reality Ecosystem	5
1.3.1 Emergent AR Applications and the Ecosystem	6
1.3.2 Placement of Wearable Cognitive Assistance in the Taxonomy	7
1.4 Thesis Overview	8
2 Gabriel Platform	9
2.1 Design Constraints	10
2.1.1 Crisp Interactive Response	10
2.1.2 Limitation of Mobile Devices	10
2.1.3 Context-sensitive Sensor Control	11
2.1.4 Graceful Degradation of Offload Services	12
2.1.5 Coarse-grain Parallelism	12
2.2 System Architecture	13
2.2.1 Overview	13
2.2.2 Use of Cloudlets	14
2.2.3 Offload Fallback Strategy	15
2.2.4 Flexible Pub-sub Backbone	16
2.2.5 Supporting Multi-device and Multi-user Use Cases	17
2.3 Implementation	18
2.3.1 Mobile front-end	18

2.3.2	Virtual Machine Ensemble	19
2.3.3	Discovery and Initialization	20
2.3.4	Limiting Queuing Delay	21
2.3.5	Protocol for Client-server Communication	22
2.3.6	Cloudlet Launcher	23
2.4	Micro Benchmark Evaluation	25
2.4.1	Experimental Setup	26
2.4.2	Gabriel Overhead	27
2.4.3	Queuing Delay Mitigation	28
2.4.4	System Performance with Multiple Cognitive VMs	29
3	Wearable Cognitive Assistance Applications	33
3.1	Application Implementation	33
3.1.1	Overview: Application Diversity and Similarity	33
3.1.2	Pool Assistant	36
3.1.3	Ping-pong Assistant	37
3.1.4	Workout Assistant	38
3.1.5	Face Assistant	39
3.1.6	Lego Assistant	40
3.1.7	Drawing Assistant	42
3.1.8	Graffiti Assistant	44
3.1.9	Sandwich Assistant	45
3.1.10	Furniture Assistant	46
3.2	Quantifying Target Latency Bounds	46
3.2.1	Relevant bounds in previous studies	47
3.2.2	Deriving bounds from physical motion	48
3.2.3	Bounds for step-by-step instructed tasks	49
3.3	Evaluating Application Performance	52
3.3.1	Experimental Setup	52
3.3.2	Response Time with WiFi Cloudlet Offloading	53
3.3.3	Time breakdown of applications	55
3.3.4	Benefits of Cloudlets	56
3.3.5	4G LTE vs. WiFi for First Hop	57

3.3.6	Effect of Better Client Hardware	58
3.3.7	Effect of Cloudlet Hardware: Varying Number of Cores	60
3.3.8	End-to-end Latency vs. User Experienced Response Time	61
3.3.9	Effect of Cloudlet Hardware: Accelerators	63
3.3.10	Summary	64
4	Speeding up Server Computation: A Black-box Multi-algorithm Approach	65
4.1	Background: Two Key Insights	66
4.1.1	Speed-accuracy Tradeoff in Computer Vision Algorithms	66
4.1.2	Temporal Locality of Algorithm Correctness	67
4.2	A Black-box Multi-algorithm Approach	68
4.2.1	The Basic Algorithm for Result Filter	69
4.2.2	Variants of Result Filter	71
4.3	Evaluation: an Offline Analysis	73
4.3.1	Experimental Approach	73
4.3.2	Performance of Multi-algorithm Approach	74
4.3.3	Varying Number of Candidate Algorithms	76
4.3.4	Effect of Threshold Value	76
4.3.5	Randomizing input image order	77
4.4	Evaluation on Gabriel	79
4.5	Applying Multi-algorithm Approach on Other Systems	81
4.6	Discussion	83
4.6.1	An Analogy	83
4.6.2	Limitations	84
5	Optimizing Energy Consumption for Wearable Cognitive Assistance	85
5.1	Profiling Power Consumption of Gabriel Applications	86
5.2	Application-specific Sensor Control: Ping-pong	87
5.2.1	Methodology	87
5.2.2	Ping-pong Sound Detection	88
5.2.3	Experimental Setup	89
5.2.4	Evaluation	90
5.3	Application-specific Sensor Control: Lego	91

5.3.1	Methodology	91
5.3.2	Estimating Step Time	93
5.3.3	Evaluation	96
5.4	Impact of Power Saving Mode on Latency and Energy	97
5.4.1	Background of Power Saving Mode	97
5.4.2	Tradeoff between Power and End-to-end Latency	97
5.5	Chapter Summary	99
6	Related Work	101
6.1	Wearable Cognitive Assistance	101
6.2	Cyber Foraging	103
6.3	Latency Sensitivity of Mobile Applications	104
6.4	Speeding up Computer Vision	104
6.4.1	Leveraging Hardware Support	104
6.4.2	Multi-algorithm Approaches	105
6.4.3	Optimization for DNNs	105
6.5	Energy Conservation on Mobile Devices	106
7	Conclusion and Future Work	107
7.1	Contributions	107
7.1.1	Identification of New Application Category	108
7.1.2	Gabriel Platform for Wearable Cognitive Assistance	108
7.1.3	Prototype Application Implementation	109
7.1.4	Empirical Study of End-to-end Latency	109
7.1.5	Multi-algorithm Approach to Speed up Server Computation	109
7.1.6	Conserving Power Consumption with Sensor Control	110
7.2	Future Work	110
7.2.1	Scaling Wearable Cognitive Assistance	110
7.2.2	Prototyping New Applications	112
7.2.3	Better Mobile Hardware/Software Support	113
	Bibliography	115

List of Figures

1.1	Hypothetical Wearable Cognitive Assistance Use Cases	2
1.2	Three Technology Advances to Support Wearable Cognitive Assistance	4
1.3	Components of a Google Glass Device	5
1.4	Round Trip Time (RTT) with Wi-Fi & 4G LTE First Hop	6
1.5	AR Ecosystem (Scaling is approximate)	7
2.1	Gabriel Architecture	14
2.2	Offload Approaches	15
2.3	Hologram Feedback Example	19
2.4	Two Level Token-based Filtering Scheme	21
2.5	Header Example for Sensor Streaming	22
2.6	Example Feedback Message	22
2.7	Cloudlet Launcher UI	25
2.8	Example API Calls between Cloudlet Launcher and Launcher Proxy	25
2.9	Trace of CPU Frequency, Response Time Changes on Google Glass	26
2.10	Time Breakdown of Mean End-to-end Latency for the <i>NULL</i> Cognitive Module	27
2.11	Response Delay of Request	28
2.12	Breakdown of Processed and Dropped Frame for OCR	28
2.13	Summary of Implemented Cognitive Modules for Evaluation	30
3.1	Pool: Extracting Symbolic Representation	36
3.2	Ping-pong: Symbolic Representation	37
3.3	Workout: Example Volumetric Template for Sit-ups	38
3.4	Lego Task	40
3.5	Lego: Symbolic Representation	41
3.6	Example State List Representation of Lego Task	42

3.7	Draw: Symbolic Representation	43
3.8	Guidance by Drawing Assistant	43
3.9	Graffiti: Example Screenshot	44
3.10	Sandwich: Symbolic Representation	45
3.11	Furniture: Symbolic Representation	45
3.12	Deriving Latency Bound for Ping-pong from Physical Motion	48
3.13	Deriving Latency Bound for Workout from Physical Motion	49
3.14	Simple Model of System Response Time and User Action/Think Time	49
3.15	User Tolerance of System Latency	51
3.16	CDF of Application Response Time	54
3.17	Breakdown of Application Response Time	55
3.18	Latency Breakdown - WiFi vs. LTE Cloudlets	57
3.19	Latency Breakdown - Client Hardware	59
3.20	Difference Between System Latency and User Experienced Response Time (UERT)	61
3.21	Comparison of System Latency and UERT	62
3.22	Improvement of UERT with Different Number of Cognitive Engine Instances (Face)	62
3.23	CDF of Latency with and without Server-side Hardware Acceleration (GPU)	63
4.1	Tradeoff Between Speed and Accuracy in Computer Vision Algorithms	66
4.2	Temporal Locality of Algorithm Correctness	67
4.3	Adapted Gabriel Architecture for Multi-algorithm Approach to Reduce Latency	68
4.4	Processing Rate for Different Algorithms	69
4.5	Correlation between Result Probability and Correctness	75
4.6	Speed-accuracy Tradeoff of Different Algorithms in Face	76
4.7	Effect of Confidence Threshold (TH) for Face	77
4.8	Multi-algorithm Approach Result for Face	78
4.9	Multi-algorithm Approach Result for Lego	78
4.10	Multi-algorithm Approach Result for Sandwich	78
4.11	Multi-algorithm Latency Breakdown	80
4.12	Processing Rate for Different Algorithms with Multiple Best-algorithm Instances	80
4.13	Effect of Having Multiple Best-algorithm Instances (Face)	81
4.14	General System Workflow to Leverage Multi-algorithm Approach	81

4.15	Speed-accuracy Tradeoff of Different Algorithms for Object Tracking	83
5.1	Example Sequence of Ping-pong Feedback over Time in Different Configurations	90
5.2	Latency Breakdown of Different Background Pinging Mechanisms in Lego . . .	99
7.1	Wearable Cognitive Assistance Use Cases for Future Work	113

List of Tables

- 2.1 Evolution of Hardware Performance (Source: adapted from Flinn [56]) 11
- 2.2 Intrinsic Delay Introduced by Gabriel 27
- 2.3 FPS and Latency of Cognitive Modules 31

- 3.1 Prototype Wearable Cognitive Assistance Applications Developed and Studied . 34
- 3.2 Latency Bounds of Assistant Applications 47
- 3.3 Demographic Statistics 50
- 3.4 Experiment Hardware Specifications 53
- 3.5 Mean Latency (ms), Varying Number of Cores 60
- 3.6 90th Percentile System Latency vs. Latency Bounds (in milliseconds) 64

- 4.1 Candidate Algorithms for Face 74
- 4.2 Candidate Algorithms for Face 75
- 4.3 Example Detection Sequence where Multi-algorithm Approach Has the Best Accuracy 79
- 4.4 Multi-algorithm Approach Result for Object Tracking 82

- 5.1 Power Consumption of Different Sensor States 86
- 5.2 Power Consumption of Ping-pong with and without Sensor Control 89
- 5.3 Pearson Correlation [30] Coefficient Matrix of Step Time among Different Users 94
- 5.4 Step Time Estimation for Lego 95
- 5.5 Power Consumption of Lego with Different Sensor Control Policies 96
- 5.6 Power Consumption and End-to-end Latency for Wearable Cognitive Assistance . 98

Acronyms

ACK Acknowledgment.

AIDL Android interface design language.

AIMD Additive increase and multiplicative decrease.

AP Access point.

API Application programming interface.

AR Augmented reality.

CDF Cumulative distribution function.

CNN Convolutional neural network.

DAG Directed acyclic graph.

DNN Deep neural network.

FPGA Field-programmable gate array.

FPS Frames per second.

GPU Graphics processing unit.

HOG Histogram of oriented gradients.

IP Internet protocol.

JSON JavaScript object notation.

LAN Local area network.

MA Multi-algorithm approach.

OCR Optical character recognition.

OEC Open edge computing initiative.

OS Operating system.

RCNN Regions with convolutional neural network.

RTT Round-trip time.

SDK Software development kit.

SIFT Scale-invariant feature transform.

SVM Support vector machine.

TCP Transmission control protocol.

TH Threshold.

UDP User datagram protocol.

UERT User experienced response time.

UPnP Universal plug and play.

VM Virtual machine.

VPN Virtual private network.

VR Virtual reality.

WAN Wide area network.

WCA Wearable cognitive assistance.

Chapter 1

Introduction

Using a mobile device to provide a user with cognitive assistance is one of the early motivations for mobile computing. As early as 1998, Loomis et al. [118] demonstrated the use of a laptop for providing walking directions for the blind. In the same year, Davies et al. [48] described a tablet-based context-sensitive guide for tourists. Since these early roots, there have been many efforts at creating assistive applications. For example, Chroma [167] uses Google Glass to provide color adjustments for the color blind. Today, every smartphone has a GPS-based navigation app.

Although valuable, these applications are constrained by today's technology from two aspects. First, a lot of applications have to sacrifice algorithmic complexity to fit all the computation into the limited processing power of a mobile device, which is especially true for highly interactive and latency sensitive applications. Second, those applications that do reach out to the cloud for better computing power (e.g. speech recognition in Siri) usually suffer from the long latency to the cloud, thus hurting the ability to deliver a fast response. This thesis targets at enabling a new genre of *wearable cognitive assistance* applications that can leverage heavy computation at very low end-to-end latency.

Figure 1.1 presents some hypothetical use cases of such applications. These are essentially a new type of augmented reality (AR) applications. In these applications, a user focuses on performing a real-life task, while some mobile devices act like a *virtual instructor* or *personal assistant* to give her prompt guidance. The guidance comes just in time after she makes any progress. The application should also quickly catch a user's mistake and provide appropriate corrections. In order to accurately understand a user's state, the applications have to leverage the enormous data captured by relevant sensors on the mobile devices, and apply huge amount of computation for interpretation. This computation is usually too heavy to be run entirely on a mobile device, which is constrained by its weight, size, and thermal dissipation.

The core contribution of this thesis is the design, implementation, and evaluation of Gabriel, an application platform that simplifies the creation of and experimentation with new wearable cognitive assistance applications. This platform factors out common functionality across all the applications such as network communication and data preprocessing. It has a flexible architecture for easy exploitation of coarse-grain parallelism within an application to improve system performance in terms of both end-to-end latency and throughput. It also provides mechanisms to

Ron was wounded in Afghanistan and is slowly recovering from traumatic brain injury. He is often unable to remember the names of friends and relatives. He also forgets to do simple daily tasks. Fortunately, his cognitive assistance system offers hope. When Ron looks at an acquaintance for a few seconds, that person's name is whispered in his ear along with additional cues to guide Ron's greeting and interactions; when he looks at his thirsty houseplant, he hears "water me"; when he looks at his dog, he hears "take me out."

(a) Disabled Veteran

Jane wants to make her first butterscotch pudding today. She starts her "Recipe" application in her smart glass, selects butterscotch from pudding menu. After making several steps correct, the Glass whispers "Good job! Now gradually whisk in one cup of cream and stir it until smooth". Jane poured one cup, stirred for ten seconds, and then stopped, waiting for the next step instruction. Having analyzed the color and texture of the cream, the glass then reminds her, "Please stir a little bit more, this is a critical step".

(c) Cooking

John has been good on his new diet for three weeks. He has been strongly advised by his doctor to lose weight, because he is on the verge of Type-2 diabetes. Today, at a dinner with friends, John's resolve has been sorely tested but he has been able to resist all temptations so far. Alas, the dessert course is his downfall. When the mouth-watering tray of delicacies is brought before him, he can hold back no longer. As he reaches for the 1200-calorie pecan pie with whipped cream, his cognitive assistance system screams in his ear and stops him cold. John has been able to stay on his diet for another day.

(e) Health and Wellness

For Sara, transitioning from her EMT training to the field is a daunting experience. Here she is with her first emergency, a middle-aged woman who has fallen to the ground after being stung by a wasp at a state fair. With her own pulse racing, amidst the distraction of the milling crowd, Sara tries to remember the protocol for allergic reaction when her cognitive assistance system says, "Check for respiratory distress." When Sara responds "present," the voice says, "epinephrine injection device – thigh." Point by point, Sara is led through her protocol until the woman's wheezing begins to subside.

(b) Medical Training

Bob just moved to Pittsburgh and bought a lot of new furniture from Ikea. Different from past experience of reading from instruction sheets, he now starts the "Assembly Assistant" smart glass application to guide him in the assembly steps by displaying picture and video tutorials. Bob feels it's much easier to read instructions from the glass and finishes everything ahead of schedule. In one of the steps, before he tries to screw a bolt to connect two pieces, the glass warns him that the bolt in his hand is not of the correct size.

(d) Furniture Assembly

On a busy travel day, a People Mover at the airport is down. It is exhibiting jerky motion and behaving in an unsafe way not seen before. Alice, the lead engineer has run out of ideas. Nothing in her extensive experience matches this situation. Sensing her stress, her cognitive assistance system suggests that she check a recent incident report from an Asian airport with the same People Mover. This proves to be a winning idea. The symptoms appear to be similar, so Alice tries the same fix: reverting a recent software upgrade on the wayside computer. Within minutes the problem is solved and the People Mover is back in service.

(f) Industrial Troubleshooting

Figure 1.1: Hypothetical Wearable Cognitive Assistance Use Cases

carefully control the energy consumption on a mobile device.

One key technique for low-latency processing in Gabriel is to offload heavy computation to a *cloudlet*. A cloudlet is an edge-located data center that is only one wireless hop away from the mobile device. Because of the network proximity, it offers low latency, high bandwidth, and low jitter for a wireless network connection to the mobile user, making it the ideal place for computation offloading. Recent work in both research community and industry have pushed cloudlets to real world deployment [35].

On top of Gabriel, I have built nine applications and have confirmed the performance, generality, and ease of use of this platform. In particular, I evaluate the performance of these applications in terms of latency across a range of offloading configurations, mobile hardware, and wireless networks, including 4G LTE. I also devise a novel black-box multi-algorithm approach that leverages temporal locality to reduce end-to-end latency of such applications by 60% to 70%, without sacrificing accuracy. Finally, based on sensor control policies specified by each application, Gabriel is able to extend the battery life of such applications by 23% to 111%.

1.1 Thesis Statement

In this thesis, I demonstrate the possibility of building computation heavy and latency sensitive cognitive assistance applications using today’s mobile and server hardware, and wireless network technology. In particular, I claim that

A new genre of latency sensitive wearable cognitive assistance applications based on one or more mobile sensors can be enabled by factoring out common functionality onto an application platform. An essential capability of this platform is to use cloudlets for computation offloading to achieve low latency. The platform simplifies the exploitation of coarse grain parallelism on cloudlets, the conservation of energy on mobile devices, and re-targeting applications for diverse mobile devices.

The main contributions of this thesis are as follows:

- I design, implement, and evaluate the Gabriel platform, which provides common functionality to support wearable cognitive assistance applications.
- I create nine wearable cognitive assistance applications on top of Gabriel using state-of-the-art algorithms. They form a benchmark suite to deeply study the performance of different cognitive assistance system configurations.
- I analyze the performance of the benchmark applications under different system configurations, and propose approaches for improving end-to-end latency and reducing power consumption on the mobile device.

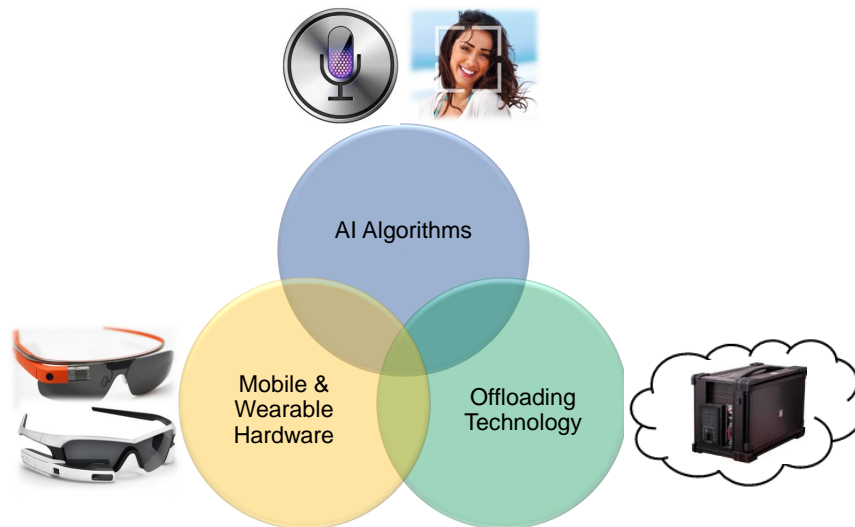


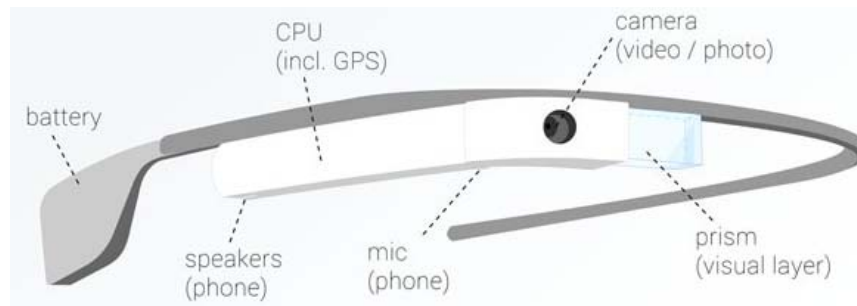
Figure 1.2: Three Technology Advances to Support Wearable Cognitive Assistance

1.2 Technology Advances for Wearable Cognitive Assistance

The possibility of using wearable devices for deep cognitive assistance (e.g., offering hints for social interaction via real-time scene analysis) was first suggested nearly a decade ago [148, 150]. However, this goal has remained unattainable until now for three reasons (Figure 1.2). First, the state of the art in many foundational technologies (such as computer vision, sensor-based activity inference, speech recognition, and language translation) is only now approaching the required speed and accuracy. For example, the recently proposed deep neural network (DNN) based machine learning approaches have dramatically improved the accuracy of speech recognition [82], object detection [146] and face recognition [152, 164]. In some of the tasks such as face recognition and image classification, computer vision has even surpassed the accuracy level of human users [79, 120]. Companies have also developed specialized hardware (e.g. TPUs [12]) and software toolkits (e.g. Tensor Flow [19]) to speed up processing such new machine learning models.

Second, the computing infrastructure for offloading compute-intensive operations from mobile devices was almost absent ten years ago. Only now, with the convergence of mobile computing and cloud computing, is this becoming convenient for use. Cloud computing services such as Amazon Cloud [24], Microsoft Azure [17], and Google Cloud [65] are among the most popular services for computation offloading. Today, with the advent of cloudlets, the connection between mobile devices and back-end servers is even tighter, resulting in better system performance. Many companies and organizations, including Nokia [134], Myoonet [127], and the Open Edge Computing initiative (OEC) [18] as a joint effort by Vodafone, Intel, Huawei, CMU, Verizon, Deutsche Telekom, T-Mobile, and Crown Castle, have been exploring the deployment of cloudlets connected through WiFi or LTE.

Third, suitable wearable hardware was not available ten years ago. Although head-up displays have been used in military and industrial applications, their unappealing style, bulkiness



Source: adapted from Missfeldt [126]

Figure 1.3: Components of a Google Glass Device

and poor level of comfort have limited widespread adoption. Only now has aesthetically elegant, product-quality hardware technology of this genre become available. Google Glass is one of the earliest and most well-known examples. It is embedded with all-round sensors to sense a user's environment, as shown in Figure 1.3. More recent devices such as Microsoft HoloLens and ODG R7 Smart Glass [135] are equipped with even better sensors for environmental tracking and 3D display.

In summary, the technology advances in AI algorithms, cloud(let) computing infrastructure, and mobile hardware have paved the way for the exploration of the new genre of wearable cognitive assistance applications. This thesis will build applications on top of the advances mentioned above, and study them from a system's perspective.

1.3 The Augmented Reality Ecosystem

The term augmented reality (AR) was coined in the early 1990s, as a contrast to the even older term *virtual reality (VR)* whose roots stretch back to the late 1960s [163]. The difference between the two is one of degree: AR only partially overlays reality with synthesized output (such as a visual image or an audible sound), whereas VR completely replaces reality with synthesized output. More recently, the term "mixed reality" has also been used. In this dissertation, I use the term "AR" to mean all forms of sensor-driven synthetic reality.

As originally conceived, AR used a sensor-rich head-mounted display that was tethered to a compute engine. This provided a deeply immersive user experience, but with the inconvenience of tethering. As AR has evolved, the tether has been replaced by a wireless network. The computation is now provided by the cloud, or a nearby compute engine. The clunky head-mounted display has been replaced by mobile devices such as smartphones, Google Cardboard, Google Glass, Microsoft HoloLens, or Vuzix Smart Glass. These changes have broadened the concept of AR.

Two constraints accentuate the tension between fast interaction and intense computation on mobile devices. First, in spite of continuing improvements in mobile hardware, there exists a

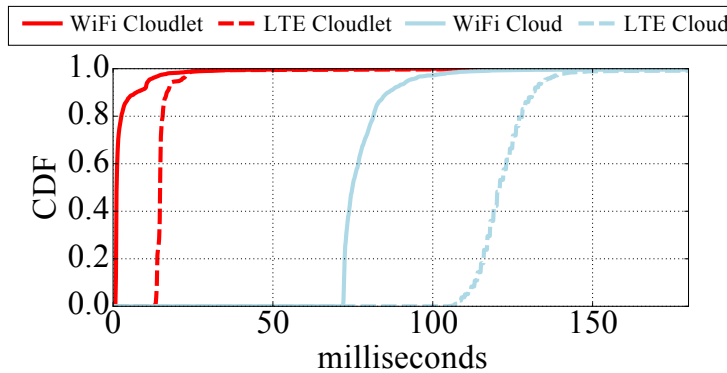


Figure 1.4: Round Trip Time (RTT) with Wi-Fi & 4G LTE First Hop

large and stubborn gap between the compute capabilities of servers and mobile devices. Because of this gap, intense computation on a mobile device is only possible through offloading to server-class hardware over a wireless network. Second, today’s wireless networks introduce significant latency into the end-to-end path of critical computation. This is shown in Figure 1.4, which presents the measured round trip time (RTT) of pings over Wi-Fi and 4G LTE (from our lab) to the cloud and to a nearby cloudlet. Sub-millisecond latencies are predicted for 5G networks [55], but these are still many years away from commercialization and widespread deployment.

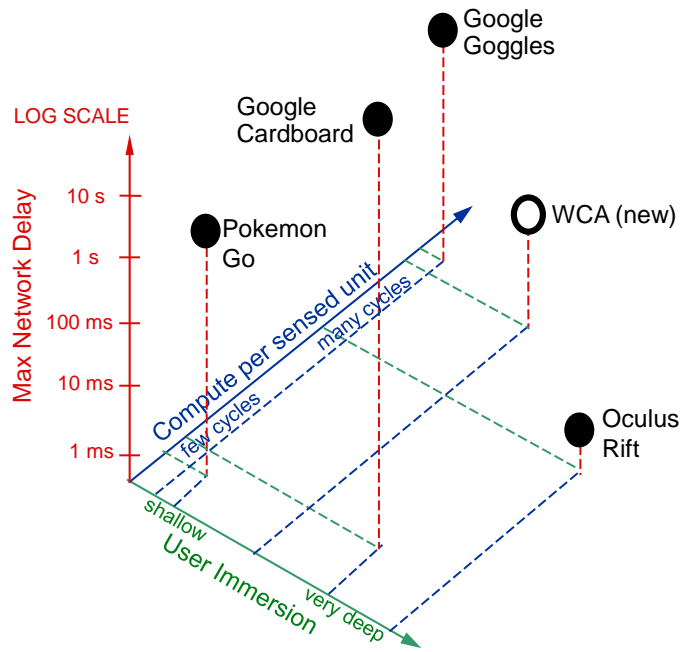
1.3.1 Emergent AR Applications and the Ecosystem

Today’s AR ecosystem has emerged in the face of these constraints. Figure 1.5 shows example AR applications and positions them according to their degree of immersion, computation complexity, and network delay tolerance. The network delay here not only includes RTT, but also the time needed for data transmission, which is dependent on network bandwidth, jitter, and data volume. Note that sometimes I directly use the hardware name (e.g. Google Cardboard) to refer to the type of applications that can be run on that device. In such cases, the user immersion should be thought of as the potential of immersion that could be enabled by such a device.

Pokémon Go involves minimal computation in the critical path of user interaction. It merely superimposes and animates an easy-to-render Pokémon on a live camera background. Periodic updates from GPS and map services are outside the critical path of user interaction, so many seconds of network delay are acceptable. The user experience is engaging, but not deeply immersive.

Google Cardboard is also light on computation but offers a much more immersive experience. For example, in a use case such as New York Times VR, it functions by selectively displaying pre-rendered output that has been prefetched. Since runtime computation is small, offloading is not necessary.

Oculus Rift offers a deeply immersive user experience using substantial computation to generate highly realistic scenes that are rapidly generated and rendered in response to sensor inputs. Only



Network delay may be affected by network RTT, bandwidth, and jitter in transmission

Figure 1.5: AR Ecosystem (Scaling is approximate)

by tethering to a high-end compute device is it possible to keep network delay low enough (a millisecond or less) to achieve the necessary depth of immersion.

Google Goggles is even more computationally demanding. On every input image, it has to perform a sequence of operations such as OCR followed by language translation, or object recognition followed by scene interpretation. This requires offloading to the cloud, which incurs significant delay (on the order of hundreds of milliseconds) with today’s networking technology. The user experience is therefore less immersive than Oculus Rift or Google Cardboard.

1.3.2 Placement of Wearable Cognitive Assistance in the Taxonomy

The class of wearable cognitive assistance applications occupies a previously-unpopulated region of the 3D space shown in Figure 1.5. Such an application continuously tracks user context by applying complex artificial intelligence (AI) algorithms, such as face recognition, object detection, speech recognition, or language translation, within a tight response time budget to live data, such as images or sound acquired by sensors on a mobile device. In other words, it combines the delay intolerance of AR applications with very high computational demands of AI applications.

Of the existing applications shown in Figure 1.5, Google Goggles is comparable to wearable cognitive assistance in terms of computational demands but is not as immersive. I characterize wearable cognitive assistance as “mildly immersive” with a network delay tolerance on the order of a few tens of milliseconds. This is large enough to allow use of today’s Wi-Fi and 4G LTE

wireless networks for untethering the mobile device, but too small for offloading to the cloud over a typical WAN — for example, Li et al [112] report that the average RTT from a random point in the Internet to its optimal cloud provider over a wired network is about 74 ms. To this must be added the first hop RTT of a wireless network, which is typically 10-15ms for today’s 4G/LTE technology.

1.4 Thesis Overview

The rest of this dissertation is organized as follows:

- In Chapter 2, I first explain the key system design constraints, and then describe the design and implementation of the Gabriel platform. Through evaluation using a series of micro-benchmarks, I show that Gabriel is able to provide very tight end-to-end latency, while allowing flexible application structure to leverage external computation resources.
- In Chapter 3, I describe nine wearable cognitive assistance applications I have built on top of Gabriel. I evaluate the performance of these realistic applications using different system configurations, explore the system design tradeoffs, and identify the performance bottleneck by studying the time breakdown of each application. In deriving a range of target latencies for each application and comparing them with actual measured latency, I show the infrastructure needed to support such applications.
- Chapter 4 describes diverse efforts to speed up the suite of applications by reducing server computation time, which is the main bottleneck as identified in Chapter 3. I propose a novel black-box multi-algorithm approach that significantly improves system response time with little impact on application accuracy. I describe how this approach can be used in Gabriel platform and evaluate it using four different applications.
- Chapter 5 focuses on reducing energy consumption of the mobile device in running wearable cognitive assistance applications. Through case studies on two prototype applications, I demonstrate that the power consumption of mobile devices can be dramatically reduced through careful sensor control. I also show the tradeoff between power consumption and application response time with system-wide power saving techniques.
- Chapter 6 discusses research and industrial efforts related to this thesis.
- Finally, in Chapter 7, I conclude this dissertation and identify important research directions for future work.

Chapter 2

Gabriel Platform

As wearable cognitive assistance is just an emerging concept, there are not many existing applications in this space, let alone open-source ones that cover a broad range of usage scenarios. As a result, I found it necessary to develop my own broad set of cognitive assistance applications covering a variety of assistive tasks. In Chapter 3, I will show that all of these applications have similar requirements and structures. Based on this, I have designed and implemented Gabriel, an underlying software platform to support these applications. Gabriel can be viewed as a PaaS (Platform as a Service) layer. It simplifies and speeds up creating new applications by factoring out complex system-level functionality that is common across many applications, including network communication and data preprocessing. Gabriel is an open-source project at <https://github.com/cmusatyalab/gabriel>.

Wearable cognitive assistance applications are usually compute intensive, and therefore cannot be run entirely on the mobile device. At the same time, end-to-end latency matters because they are human-in-the-loop applications that deeply engage user attention. As a result, a key design choice in Gabriel is to offload heavy computation to a cloudlet to achieve both high compute capability and low end-to-end latency. On a cloudlet, Gabriel provides a pub-sub backbone that simplifies the exploitation of coarse-grain parallelism. By running different cognitive modules inside different virtual machines (VMs), an application could easily scale out to achieve higher throughput and even lower latency (explained in Chapter 4). In addition, Gabriel has provided standard mechanisms to receive commands from applications for careful control of wearable sensors. This helps to reduce the energy consumption on the mobile device without affecting user experience.

In this chapter, I will first list several key design considerations and constraints in building Gabriel. Based on this, I will introduce the design of Gabriel architecture, and describe the implementation details. Finally, I will evaluate the performance of Gabriel using several micro benchmarks. A more thorough evaluation using realistic wearable cognitive assistance applications is left to the next Chapter.

2.1 Design Constraints

2.1.1 Crisp Interactive Response

Timely responses are a critical aspect of cognitive assistance applications. If a guidance is delayed for too long, it may no longer be relevant (e.g., in a face recognition application, the recognized person in the scene may have already left before the slow guidance arrives). Even if a late guidance can provide useful information, the user may be annoyed by the long delay. In this dissertation, I call such timeliness required by cognitive assistance as “crispness”, and the associated responses as “crisp responses”.

How fast is fast enough? Although answering this question requires careful study of the user and task, it is best if our system can match or even surpass the human speed. In the example of face detection and recognition, Lewis et al. [111] report that even under hostile conditions such as low lighting and deliberately distorted optics, human subjects take less than 700 milliseconds to determine the absence of faces in a scene. For face recognition under normal lighting conditions, experimental results on human subjects by Ramon et al. [144] show that it takes only 370 milliseconds for a human to realize a face is familiar and up to 620 milliseconds to confirm unfamiliarity. Similarly, for speech recognition, Agus et al. [21] report that human subjects recognize short target phrases within 300 to 450 ms.

The Gabriel platform is designed to make it easy for applications built on top of it to meet these numbers. Therefore, the latency overhead of the system infrastructure should be as small as possible, ideally an order of magnitude smaller than the few hundreds of millisecond latency budget for many applications. Thus, I set the goal for the latency of Gabriel infrastructure to be a few tens of millisecond.

2.1.2 Limitation of Mobile Devices

Mobile devices are always resource poor relative to fixed infrastructure in computing power. Table 2.1, adapted from Flinn [56], illustrates the consistent large gap in the processing power of typical server and mobile device hardware over nearly 20 years. This stubborn gap reflects the fact that although mobile devices are becoming more powerful thanks to Moore’s law, a server of comparable vintage is always much more powerful. Moreover, the design of mobile devices has always been constrained by the requirement of being light weight and small, having sufficient battery life, comfortable to wear and use, and tolerable heat dissipation, which makes it harder to further increase the processing power.

The large gap in the processing capabilities of wearable and server hardware directly translates to the difference in response time a user gets. For example, I measured the response time of a representative cognitive assistance application (optical character recognition (OCR) using Tesseract-OCR package [160]). When the entire application runs on a Google Glass and involves no network communication, it takes more than 10 seconds to process one test image. However, if the OCR processing is offloaded to a nearby compute server (a Dell Optiplex 9010 desktop with an Intel® Core™ i7 4-core processor and 32GB of memory) through WiFi, it takes

Year	Typical Server		Typical Handheld or Wearable	
	Processor	Speed	Device	Processor Speed
1997	Pentium® II	266 MHz	Palm Pilot	16 MHz
2002	Itanium®	1 GHz	Blackberry 5810	133 MHz
2007	Intel® Core™ 2	9.6 GHz (4 cores)	Apple iPhone	412 MHz
2011	Intel® Xeon® X5	32 GHz (2x6 cores)	Samsung Galaxy S2	2.4 GHz (2 cores)
2013	Intel® Xeon® E5-2697v2	64 GHz (2x12 cores)	Samsung Galaxy S4	6.4 GHz (4 cores)
			Google Glass	2.4 GHz (2 cores)
2016	Intel® Xeon® E5-2698v4	88.0 GHz (2x20 cores)	Samsung Galaxy S7	7.5 GHz (4 cores)
			HoloLens	4.16 GHz (4 cores)
2017	Intel® Xeon® Gold 6148	96.0 GHz (2x20 cores)	Pixel 2	9.4 GHz (4 cores)

Table 2.1: Evolution of Hardware Performance (Source: adapted from Flinn [56])

only about one second to process, giving almost an order of magnitude improvement in system response time.

Besides the limitation of computing power, energy consumption is another key constraint in designing applications for mobile devices, especially wearable devices, as the limited size and weight incurs strict constraints on battery capacity. For example, a Google Glass battery can only support roughly 45 minutes of continuous video recording. This number is not likely to improve much in the near future, since the battery technology does not follow Moore’s Law. Fortunately, because of the reduction of CPU load, computation offloading has also offered an opportunity to reduce energy consumption on a mobile device. In the OCR example above, it consumes about 0.89 Watts when offloading, compared to 1.22 Watts when running native. For the reasons above, computation offloading is essential in Gabriel for fast processing and efficient energy usage.

2.1.3 Context-sensitive Sensor Control

Even with computation offloading, the continuous sensor reading and data streaming would consume significant energy on the wearable device. Further improvements in battery life and usability are possible through careful control of the wearable sensor and network usage. For example, consider a user who continuously runs the face recognition assistance application in her daily life, but is now taking a nap on a chair. While she is asleep, the wearable device does not have to capture video and stream it for cognitive assistance. When she wakes up of her own ac-

cord, processing for cognitive assistance should resume promptly. Classic activity recognition mechanisms based on body-worn sensor data [130] could be applied here to reliably distinguish between the user’s sleeping and waking states. This type of high-level knowledge of user context could be used to benefit many applications.

On top of this, more fine-grained control of sensor and network usage can be achieved by leveraging application-specific knowledge. For example, in an application to assist Ikea kit assembly, a user needs at least several seconds to several minutes to perform each assembly step. The wearable sensors and networking interface could be turned off during these periods, and later turned on when the user is ready for the next step. Of course, it is not possible to know in advance when the user will be ready for the next step, thus the application may have to make an estimation of the duration of each step. It is also possible to keep low-power sensors (e.g. accelerometer) always on to detect whether the user is ready or not, and turn all sensors back on for full analysis when the user is ready. Regardless of the strategy taken by each application, the Gabriel platform should provide a simple and standard way for applications to specify their sensor data needs, and to control the sensors appropriately.

2.1.4 Graceful Degradation of Offload Services

What does a user do if a network failure, server failure, power failure, or other disruption makes offload impossible? While such failures will hopefully be rare, they cannot be ignored in an assistive system. Simply falling back on standalone execution on his wearable device will hurt user experience. As the results in Section 2.1.2 suggest, it will negatively impact crispness of interaction and battery life. Under these difficult circumstances, the user may be willing to sacrifice some other attribute of his cognitive assistance service in order to obtain crisp interaction and longer battery life. For example, an assistive system for face detection and recognition may switch to a mode in which it only recognizes bigger faces, which usually correspond to people who are closer to the user; when offloading becomes possible again, the system can revert to recognizing all faces within sight. Of course, the user will need to be alerted to these transitions so that his expectations are set appropriately.

This general theme of trading off *application-specific fidelity of execution* for response time and battery life was explored by Noble [133], Flinn [57], Narayanan [129], and others in the context of the Odyssey system. Their approach of creating an interface for applications to query system resource state and to leave behind notification triggers is applicable here. However, assistive applications have very different characteristics from applications such as streaming video and web browsing that were explored by Odyssey.

2.1.5 Coarse-grain Parallelism

Many tasks for cognitive assistance can be assembled from multiple building blocks. These building blocks may run independently to analyze different aspects of a user’s context, or some of them may depend on others’ processed results for further processing. For example, in an application that offers identity information of people nearby, a face recognizer may use the results

of a face detector as inputs. At the same time, an OCR based name-card detector could be run in parallel to confirm identity. These different software building blocks are natural units to be run in parallel. In this dissertation, I call this kind of parallelism coarse-grain parallelism, as each parallel unit is running an independent task, and can be treated as a blackbox.

It is crucial for a cognitive assistance system to simplify the exploitation of such coarse-grain parallelism. The first reason is it makes it easier to leverage existing work. While a wide range of software building blocks for different cognitive assistance applications exist today, such as face recognition [171], natural language translation [31], and OCR [160], they are usually written in a variety of programming languages and use diverse runtime systems. In their entirety, these existing bodies of work represent many hundreds to thousands of person years of effort by experts in each domain. Only through coarse-grain parallelism can we easily reuse this large body of existing code. Second, parallelism helps performance. By running multiple instances of the same processing component, we can easily increase the system throughput. In Chapter 4, I will also show how such parallelism can help to reduce system latency.

Interestingly, the coarse-grain parallelism also resembles the processing model of a human brain. Human cognition involves the synthesis of outputs from real-time analytics on multiple sensor stream inputs. A human conversation, for example, involves understanding the language content and deep semantics of the words, sensing the tone in which they are spoken, interpreting the facial expressions with which they are spoken, and observing the body language and gestures that accompany them. There is substantial evidence [141] that human brains achieve such impressive real-time processing by employing different neural circuits in parallel and then combining their outputs. Coarse-grain parallelism is thus also at the heart of human cognition.

2.2 System Architecture

The high-level design of Gabriel is strongly influenced by the constraints described in the previous section. In this section, I will first present an overview of Gabriel architecture, and then describe how each key component helps to meet the constraints. Implementation details of the Gabriel system will be described in Section 2.3.

2.2.1 Overview

Figure 2.1 illustrates the overview of Gabriel’s architecture. A front-end on a wearable device does basic preprocessing, and then streams sensor data (e.g. video and audio streams, accelerometer data, GPS information, etc.) over a wireless network to the Gabriel back-end. An ensemble of virtual machines (VMs) on the back-end handles these sensor streams. The *Control VM* is responsible for all Gabriel interactions with the wearable device, as well as common functionality for all cognitive applications such as decoding of the sensor stream. Gabriel uses a publish-subscribe (PubSub) mechanism to distribute the decoded sensor streams to each *Cognitive VM* that is in charge of one cognitive processing component. The outputs of the Cognitive VMs are sent to a single *User Guidance VM* that integrates these outputs and performs higher-level

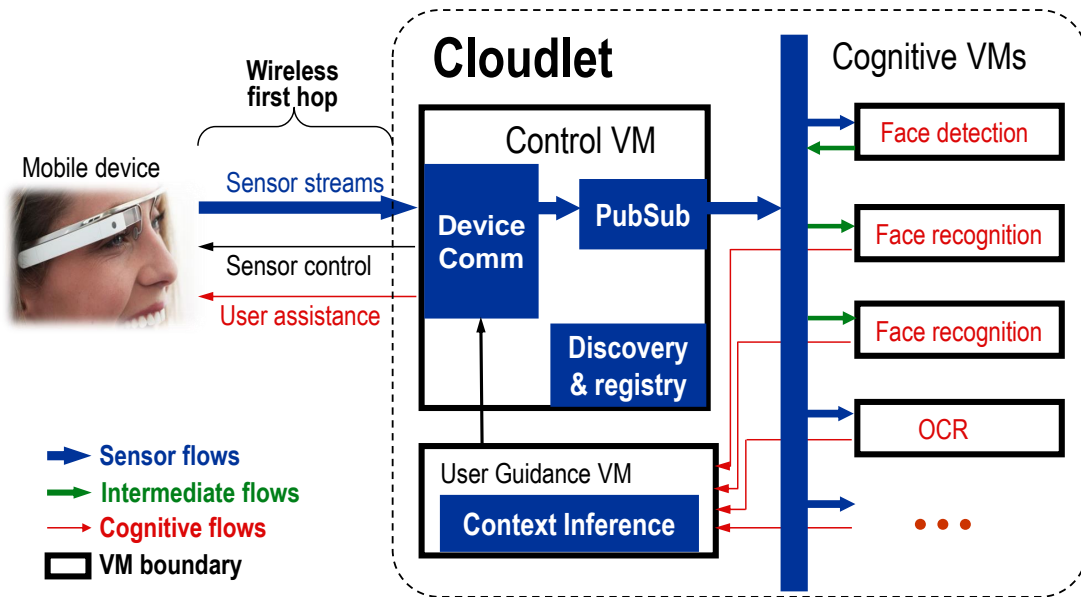


Figure 2.1: Gabriel Architecture

application-specific processing. From time to time, this triggers output for user assistance and transmits the guidance back to the wearable device via the Control VM. The guidance is sent using JSON format, including a spoken audio whispered into the user’s ears, and/or an image displayed on the screen of the mobile device.

Context-sensitive sensor control (Section 2.1.3) is addressed in Gabriel by having a context inference module inside of the User Guidance VM. This module can detect generic user context such as whether she is stationary or moving. It could also leverage the results from Cognitive VMs to detect application specific context such as the progress and speed in an assembly task. Through JSON messages sent back to the user, the context inference module could embed control commands to the wearable device, such as turning on/off a sensor or adjusting the data capture rate. It could also schedule a sensor control operation in a future time, such as to turn on a particular sensor 10 seconds later.

2.2.2 Use of Cloudlets

Gabriel faces the difficult problem of simultaneously satisfying the need for crisp, low-latency interaction (Section 2.1.1) and the need for offloading computation from a wearable device (Section 2.1.2). The obvious solution of using commercial cloud services over a WAN is unsatisfactory because the round trip time (RTT) is too long. Li et al. [112] report that the average RTT from 260 global vantage points to their optimal Amazon EC2 instances is nearly 74 ms, and a wireless first hop would add to this amount. This makes it virtually impossible to meet the latency goal of a few tens of milliseconds for our infrastructure, as set up in Section 2.1.1.

Gabriel achieves low-latency processing by offloading to cloudlets [150]. A cloudlet is a powerful, well-connected small datacenter that is just one wireless hop away. It can be viewed

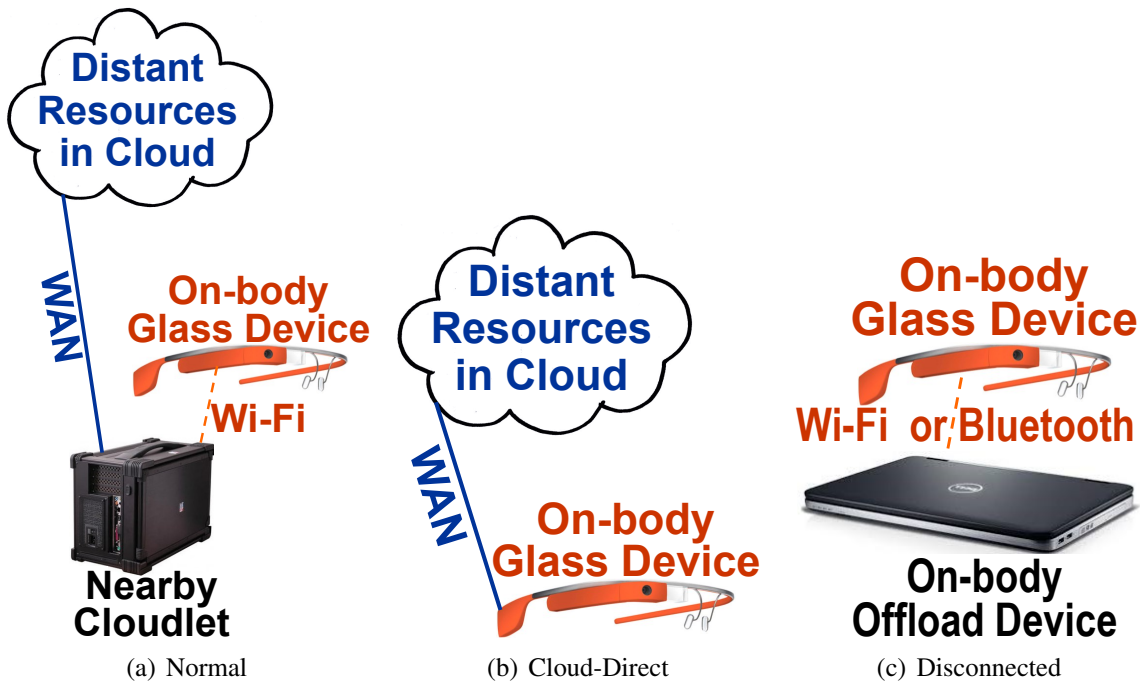


Figure 2.2: Offload Approaches

as a “data center in a box” whose goal is to “bring the cloud closer.” The low latency and high bandwidth access to a cloudlet makes it an ideal offload site for cognitive assistance. Recently, cloudlets have attracted great attention from industry for real deployments, and they may be variously known as micro data centers [69], fog [32], or mobile edge cloud [35]. We use the term “cloudlet” throughout this thesis to refer to such an edge-located data center.

The cloudlet is not used to fully replace the cloud. Instead, the mobile device, the cloudlet, and the cloud form a 3-tier hierarchy [73]. Figure 2.2(a) illustrates this ideal scenario: the cloudlet is in charge of all the cognitive processing and latency-sensitive device-server interactions to offer timely assistance to a user, while the cloud has a centralized view of software version control and usage logging. The periodic information exchange between a cloudlet and the cloud is thus outside of the critical path of assistive applications.

2.2.3 Offload Fallback Strategy

When no suitable cloudlet is available (Section 2.1.4), the obvious fallback is to offload directly to the cloud as shown in Figure 2.2(b). This incurs the WAN latency and bandwidth issues that were avoided with cloudlets. Since RTT and bandwidth are the issues rather than processing capacity, application-specific reduction of fidelity must aim for less frequent synchronous use of the cloud. For example, a vision-based indoor navigation application can increase its use of dead reckoning and reduce how often it uses cloud-based scene recognition. This may hurt accuracy, but the timeliness of guidance can be preserved. When a suitable cloudlet becomes available, normal offloading can be resumed. To correctly set user expectations, the system can use audible

or visual signals to indicate fidelity transitions. Some hysteresis will be needed to ensure that the transitions do not occur too frequently.

An even more aggressive fallback approach is needed when the Internet is inaccessible. If the mobile device is powerful enough, such as a modern smartphone, the cognitive processing may be able to run directly on the device. However, some wearable devices are not capable of performing all cognitive processing natively, as shown by the results in Section 2.1.2. To handle these extreme situations, I assume that the user is willing to carry a device such as a laptop or a netbook that can serve as an offload device. Powerful smartphones could also be used for offloading. The preferred network connectivity is Wi-Fi with the fallback device operating in AP mode, since it offers good bandwidth without requiring any infrastructure. For fallback devices that do not support AP mode Wi-Fi, a lower-bandwidth alternative is Bluetooth. Figure 2.2(c) illustrates offloading while disconnected.

How large and heavy a device to carry as fallback is a matter of user preference. A larger and heavier fallback device can support applications at higher fidelity and thus provide a better user experience. With a higher-capacity battery, it is also likely to be able to support a longer period of disconnected operation than a smaller and lighter device. However, running applications at higher fidelity shortens battery life. One can view the inconvenience of carrying an offload device as an insurance premium. A user who is confident that Internet access will always be available, or who is willing to tolerate temporary loss of cognitive assistance services, can choose not to carry a fallback offload device.

2.2.4 Flexible Pub-sub Backbone

The pub-sub backbone in Gabriel enables the exploitation of coarse grain parallelism, addressing the constraint in Section 2.1.5. By encapsulating different cognitive modules into VMs, a cloudlet can be scaled out by simply adding more independent processing units, leading eventually to an internally-networked cluster structure. This backbone helps application developers to quickly prototype and experiment new applications. For example, a conference organizer may want to build an assistive application that provides each attendee's basic information, such as institution and specialty. This could be easily achieved by plug in public available face detection and recognition modules as Cognitive VMs to the pub-sub backbone. The only additional development needed is to retrieve user information in the User Guidance VM, based on identity returned by the face recognizer. In this sense, the cognitive modules can be seen as micro-services that are reusable by different applications.

It is important to note that the Cognitive VMs don't have to run independently of each other. An application running on top of Gabriel could have arbitrary dependencies among different components, as long as they form a directed acyclic graph (DAG). Some of the Cognitive VMs may receive appropriate sensor streams directly from the Control VM, while others may take the results from other components for further processing. For example, in the example illustrated in Figure 2.1, the face detection Cognitive VM takes the video stream from the Control VM as input. It then publishes the cropped face image stream back to the pub-sub backbone (indicated by the yellow arrow). The face recognition VMs then use these as the starting point to identify the

person. Since face recognition may be slower than face detection, two identical face recognizers are run concurrently to improve overall throughput.

Gabriel achieves such flexible pub-sub infrastructure by having a centralized registry and discovery server inside the Control VM. When the Cognitive VMs or the Control VM have data streams to publish (e.g. original video stream or cropped face image stream), they register them at this registry. Other VMs could then simply subscribe to these streams for further processing.

2.2.5 Supporting Multi-device and Multi-user Use Cases

Normally, a wearable cognitive assistance application requires only one wearable device for a single user. Both the application inputs (sensor data streams) and outputs (feedback to the user) are taken from or shown to the same device. However, some tasks may require inputs from multiple devices for better assistance. For example, a navigation assistant may prefer to use the IMU sensor data on a smart wristband for dead reckoning, while using the video streams from the smart glass for landmark detection. In addition, separating input and output devices may be beneficial. For example, users may prefer to use a smart TV or a mounted smartphone to display image feedback for better readability, while still be able to use the smartglass to capture first-person video for analysis.

Gabriel supports such multi-device use cases by allowing multiple client devices to connect to the same Control VM. Upon connection, each client provides its device type (e.g. phone, glass, TV, etc.). The Cognitive VMs could then specify both device type and desired sensor stream for analysis. The feedback message could also be delivered to the most suitable device. Note that sometimes the device type is not a strict requirement, but rather a preference. For example, the image feedback should ideally be shown in a smart TV. If no TV is currently connected, Gabriel can fallback to display on the phone or on the glass. This preference order can be specified by the application on initialization.

A more complex scenario would have multiple users involved in one application. Gabriel currently does not support such use cases. This would be possible, however, if Gabriel is further extended to have appropriate user management, state management for different users, and information sharing among users. This extension will be explained more in the discussion of future work in Chapter 7. Note, however, that this does not mean that only one user could use Gabriel on one cloudlet. For each user, Gabriel provisions one Control VM, one User Guidance VM, and a set of Cognitive VMs, so that different users will not interfere with each other. The optimization of cloudlet resource utilization by possibly sharing Cognitive VMs among users is also part of the future work.

2.3 Implementation

2.3.1 Mobile front-end

The mobile front-end of Gabriel does very basic preprocessing, such as compression, before the sensor data is streamed to the Gabriel back-end through a TCP connection. In Gabriel, a mobile device can transmit data using either WiFi or cellular network, although some devices only support WiFi due to their hardware limitation. After connecting to the Gabriel back-end, the front-end streams video, audio, and accelerometer readings over the network to the Control VM. To achieve low network latency, I have tested both TCP and UDP. The experiments showed negligible difference in response times between TCP and UDP over Wi-Fi. Since the use of UDP adds complexity for maintaining reliable data transmission, Gabriel uses TCP for each sensor stream.

So far, I have implemented both Android and Windows clients for Gabriel front-end, which could be run on many modern wearable or mobile devices, including Nexus 6 phone, Google Glass [64], Microsoft HoloLens [123], Vuzix M100 smartglass [177], and ODG R7 smart-glass [135]. Because the bulk of an application's implementation lies within the Gabriel back-end, I also expect that it will be simple to port the front-end to other wearable devices with different development platform, including iOS.

Android Client

The current Android client is targeting Android 4.4.2, which is the highest Android version supported by some of the wearable devices, such as Google Glass, Vuzix M100, and ODG R7. The portability of Android makes it easy to run the Gabriel front-end software on compatible devices without any change in code. For Google Glass, I am also enabling Google Glass-specific features such as application control via voice. Thus, users can launch the Gabriel front-end application using voice. Once a guidance message is received, the Android devices display image guidance as bitmaps on the screen and audio guidance as synthesized speech using the Android text-to-speech API.

Windows Client

The general Windows client is a Universal Windows Platform (UWP) application that runs on Windows 10. It does the exact same preprocessing on the device as the Android client, and uses the same network protocol to communicate with the Gabriel back-end. It also supports the same set of feedback types. Although only tested with HoloLens, this Windows client should be able to run on all Windows 10 devices, including Windows phones and desktop PCs.

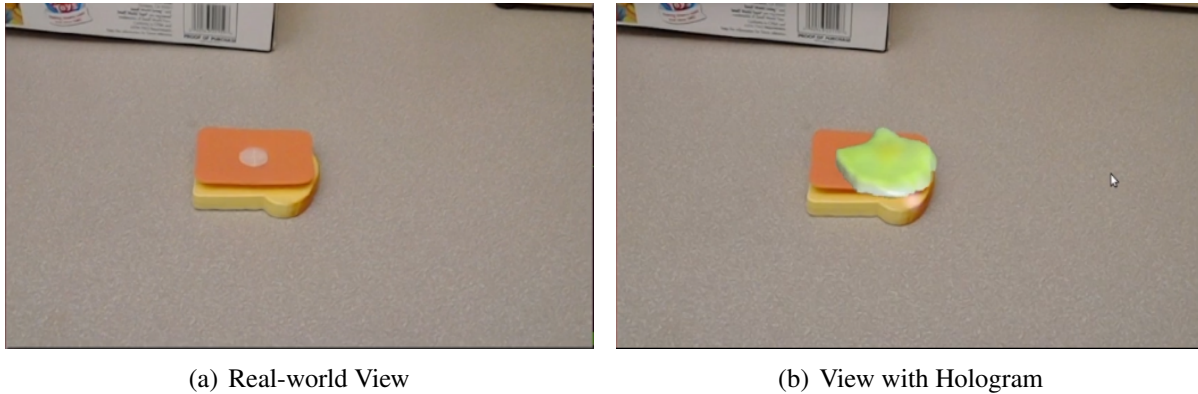


Figure 2.3: Hologram Feedback Example

HoloLens Client

A powerful feature of HoloLens is the capability of placing virtual objects in the form of holograms in the real world. This could be leveraged to provide a new type of hologram-based user guidance that is useful in many tasks. Therefore, I have created another Windows client, specifically targeting the HoloLens device. This client software leverages HoloLens’s ability for spatial understanding and hologram placement, and is built on top of the Unity Engine [172] for interaction with 3D objects. Before running the application, 3D models of the guidance objects must be created, possibly through 3D scanning. Figure 2.3 shows an example of hologram feedback in an sandwich assembly task, which will be explained in detail in Chapter 3. The left image (Figure 2.3(a)) shows the original image captured by the user’s HoloLens, and the right image (Figure 2.3(b)) shows the virtual lettuce on top of the ham, instructing the user of the next step.

Currently, the HoloLens client can only capture images at two frames per second. This is because HoloLens needs to use the camera to continuously track the movement of the device, thus each application can only occasionally have access to the camera to take snapshots. To capture images at high framerate like in other Gabriel front-ends would require system changes to share camera resources between multiple processes. Because of this limitation, this HoloLens-based front-end will not be evaluated in later sections of this thesis.

2.3.2 Virtual Machine Ensemble

As explained earlier in Section 2.1.5, an important goal of Gabriel is to reuse existing software of diverse running environments to the extent possible. Gabriel achieves this by encapsulating each cognitive processing unit (complete with its operating system, dynamically linked libraries, supporting tool chains and applications, configuration files and data sets) in its own virtual machine (VM). In this way, any existing software, no matter in Windows or Linux, can easily fit into the Gabriel platform.

However, for all of the Gabriel components to work together, we do need each Cognitive VM

to follow a common communication paradigm. To this end, if any existing software building block is not following the paradigm, the system would require a small amount of glue logic to be written for each cognitive module. This code can be thought of as a wrapper around the cognitive module, transforming the inputs and outputs to work with our system. It pulls data from the input streams and presents it in a manner needed by the cognitive module (e.g., in the right image format, or as a file). It also extracts results from the module-specific form and transmits it as a processed data stream. This wrapper is also responsible for discovering and connecting to the PubSub system, subscribing to the required sensor streams, and publishing the outputs. Finally, the wrapper ensures that the cognitive module processes each data item (e.g., by signaling a continuously running module that new data is available, or relaunching the module for each item). All of these functions are simple to implement, and tend to follow straightforward patterns. However, as they are tailored to specific cognitive modules, they are expected to be packaged with the modules in the Cognitive VMs.

The use of VMs offers additional benefits. First, VM provides strong isolation across multiple cognitive applications. Second, existing VM migration technology makes it easy for Gabriel to scale out and scale up to fully leverage the processing units available in the internally-networked cloudlet cluster. Lighter weight encapsulation such as Docker containers [122] is also possible in the Gabriel implementation, but it incurs restrictions in terms of OS choices, and does not change the system performance significantly. Therefore, for most of the thesis, I assume the use of VMs.

2.3.3 Discovery and Initialization

The Gabriel back-end consists of many software components in multiple VMs working together. The key to making this composition work is a mechanism for the different components to easily discover and connect with each other. In my implementation, Gabriel uses UPnP protocol for discovery among VMs.

Upon initialization, the Control VM is launched first, which hosts a UPnP server, as well as a registry that keeps track of available streams and connected VMs. The User Guidance VM and the Cognitive VMs then start by performing a UPnP query to discover and connect to the Control VM. After this, they register their processed data streams to the Control VM and subscribe to the desired input streams through the PubSub system. At a lower level, this establishes direct TCP connections between any publishers and subscribers.

The different VMs are connected together on a private virtual network using VLAN. This private network can span machine boundaries, allowing sizable cloudlets to be used. The system isolates each user's data from others' by creating a VLAN per user. This guarantees that VMs belonging to different users cannot connect to each other. It also ensures user privacy, which is crucial in these applications since the first-person images as well as other sensor data are highly personal. The Control VM is connected to both the private network and the regular LAN, with a public IP address. The latter is used to communicate with the mobile device.

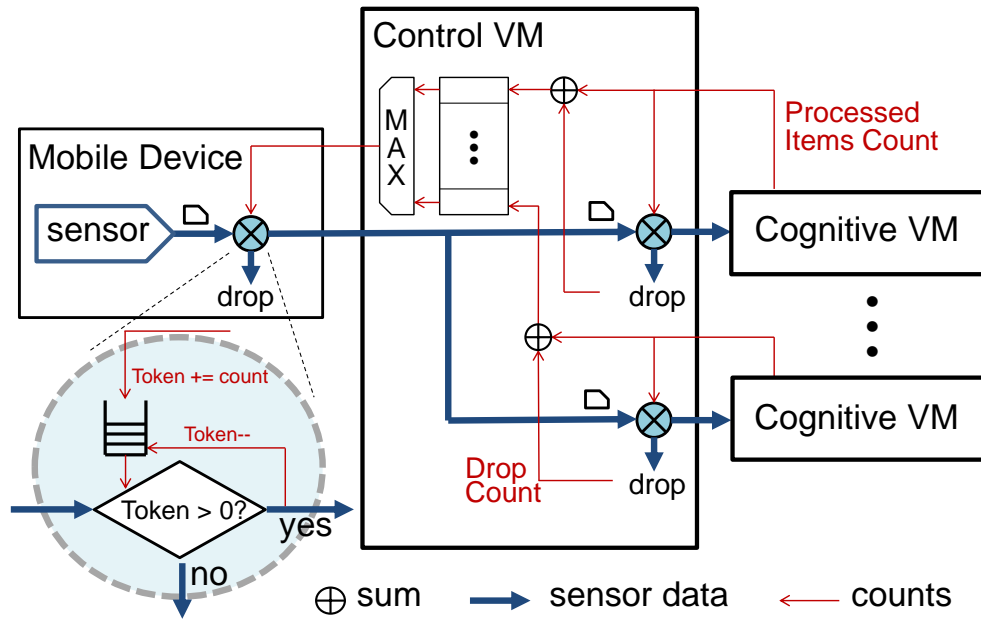


Figure 2.4: Two Level Token-based Filtering Scheme

2.3.4 Limiting Queuing Delay

In the Gabriel architecture, a set of components communicate using network connections. Each communication hop involves a traversal of the networking stacks, and can involve several queues in the applications and guest OSs, over which one has little control. The application and network buffers can be large, and cause many items to be queued up, increasing latency. To minimize queuing, one needs to ensure that the data ingress rate never exceeds the bottleneck throughput, whose location and value can vary dramatically over time. The variation arises from fluctuations in the available network bandwidth between the Glass device and cloudlet, and from the dependence of processing times of cognitive engines on data content.

To limit the total number of data items in flight at a given time in Gabriel, I have designed and implemented an application-level, end-to-end flow control system. A token-bucket filter is used to limit ingress of items for each data stream at the mobile device, using returned counts of completed items exiting the system to replenish tokens. This provides a strong guarantee on the number of data items in the end-to-end processing pipeline, limits any queuing, and automatically adjusts ingress data rate (frame rates) as network bandwidth or processing times change.

To handle multiple cognitive engines with different processing throughputs, I add a second level of filtering at each Cognitive VM (Figure 2.4). This achieves per-engine rate adaptation while minimizing queuing latency. Counts of the items completed or dropped at each engine are reported to and stored in the Control VM. The maximum of these values are fed back to the source filter, so it can allow in items as fast as the fastest cognitive engine, while limiting queued items at the slower ones. The number of tokens on the mobile device corresponds to the number of items in flight. A small token count minimizes latency at the expense of throughput and resource utilization, while larger counts sacrifice latency for throughput.

```

{
  "frame_id": 117,
  "device_type" : "glass",
  "sensor_type" : "video",
  "time_captured" : 1449460401652,
}

```

Figure 2.5: Header Example for Sensor Streaming

```

{
  "frame_id": 117,
  "input_device_type" : "glass",
  "output_device_type" : "phone",
  "sensor_type" : "video",
  "result" : {
    "speech" : "Great! Now put the lettuce on top of ham",
    "image" : "/9j/4AAQSk...iigD//2Q==",
  }
  "control" : {
    "sensor_type" : "video",
    "img_width" : 1280,
    "img_height" : 720,
  }
  "time_captured" : 1449460401652,
}

```

Figure 2.6: Example Feedback Message

Although implemented as part of Gabriel, this token-based flow control mechanism is a general approach that could be applied to other systems to get rid of system queuing delay. This is especially helpful for latency-sensitive applications, as well as dealing with multiple processing components with different throughput. The token count provides a simple parameter to trade latency for throughput.

2.3.5 Protocol for Client-server Communication

This section describes in detail the packet format used for communication between the Gabriel front-end and the back-end. Figure 2.5 shows an example header that precedes a video frame being sent. This JSON header is relatively simple. The first field is `frame_id`, which shows the frame's relative position in the stream. The second and third field illustrates the type of data that will be followed by this header, so that it could be distributed to the appropriate cognitive component. Finally, the `time_captured` field is a timestamp used for measurements and

performance analysis.

The feedback message has a more complex structure (Figure 2.6). It inherits the `frame_id` field from the input header, and uses this field as an ACK. This number is essential for the client to correctly replenish tokens for end-to-end flow control. Next, two device types are specified. The `input_device_type` is inherited from the input header. Along with the `sensor_type` field, it helps to fill tokens for the right sensor stream on the right device. The `output_device_type` determines where the guidance needs to be shown. As mentioned in Section 2.2.5, this could be different from the input device. The timestamp of data being streamed is also kept in this JSON structure. The difference between this timestamp and the time the feedback message is received is the end-to-end latency.

The `result` field contains the actual feedback information, in various formats. As the example shows, it could contain a speech string to be spoken out by the output device, or an image to be shown on the screen. The image is in JPEG format and then encoded into Base64 strings [16] for JSON compatibility. Other supported formats (not shown in the example) include animation, which is a group of images, and video, which contains an URL pointing to a playable source. If the output device is HoloLens, hologram feedback can also be included by specifying the hologram model and location (in terms of direction and distance relative to the camera). Occasionally, the feedback message also contains the `control` field for sensor control. The example in Figure 2.6 uses this field to set the camera resolution to 720P.

2.3.6 Cloudlet Launcher

So far, the Gabriel front-end software assumes the knowledge of the IP address of the Control VM before the application starts. In practice, however, a number of steps need to occur in order to initiate utilization of cloudlet resources before application running.

First, the client must perform discovery of available cloudlets, then an association must occur between the selected cloudlet and the client. Once an association is established, the client can request the provisioning of back-end resources, such as VMs and containers, before it connects to the Control VM. Since the cloudlet may not be able to assign every Control VM a public IP address [95], a VPN channel is then needed to be set up between the client and the cloudlet for appropriate network communication. Finally, if a user moves as she uses the application, an automatic VM handoff [75] might happen to ensure the user always associated with the optimal cloudlet.

I have built an Android service which is referred to as the Cloudlet Launcher, which interfaces with these capabilities. It factors out the complexity of associating with cloudlets from any applications, including the Gabriel front-end software. Although this implementation is tied specifically to Android running environment, the high level design could also be applied to other mobile platforms, such as iOS, Windows, etc.

The Cloudlet Launcher runs as a background service in Android. It could be *bound* by any other Android applications through the Android *intent*. After the connection to Cloudlet Launcher is instantiated, an application can use the Android Interface Design Language (AIDL)

APIs exposed by the service for cloudlet operations. There could be at most one Cloudlet Launcher running in an Android device, while multiple applications may be bound to it concurrently.

The Cloudlet Launcher has been published in Google Play for internal testing [4]. The current VPN management has been leveraging a third-party application called *OpenVPN for Android* that could also be downloaded from Google Play store. Should a need arise to extend or modify the VPN software, Cloudlet Launcher may implement its own version of VPN client in future releases.

Launcher API

The current functionality exposed by the Cloudlet Launcher API is listed below. The goal is to provide an extensible framework that could encapsulate all the functionality required to serve as the connective tissue between the mobile application and the cloudlet. This set may grow to include more sophisticated cloudlet discovery or registration, various authentication mechanisms, metering or usage monitoring, and migration between cloudlets.

- `boolean isServiceReady()` Goes through a checklist to see if Cloudlet Launcher service is ready to use, such as whether all dependent packages are installed, whether user profile has been set, whether network condition is good, etc.
- `void findCloudlet(String appId)` Initiates cloudlet discovery. Once a suitable cloudlet is found, it posts a request to the cloudlet to launch the required VM instance(s) according to the application ID (`appId`). Once the IP address of the Control VM is returned, VPN connection is established, if not already used by other applications.
- `void disconnectCloudlet(String appId)` Shuts down VPN connection, and destroys associated VM instance(s) through a request to the cloudlet.
- `void registerCallback(ICloudletServiceCallback callback)`
- `void unregisterCallback(ICloudletServiceCallback callback)`

As shown from the last two APIs above, each application can register a callback to Cloudlet Launcher to receive any updates regarding cloudlet states. Currently, each callback should implement three functions, as shown below.

- `void amReady()` Called once Cloudlet Launcher service is ready to accept any API calls.
- `void newServerIP(String ip)` Update the Control VM's IP to each application. This could be called when a cloudlet discovery is finished, or when a VM has been migrated to a new cloudlet.
- `void message(String msg)` Forward any other state information to the application, including debugging messages.

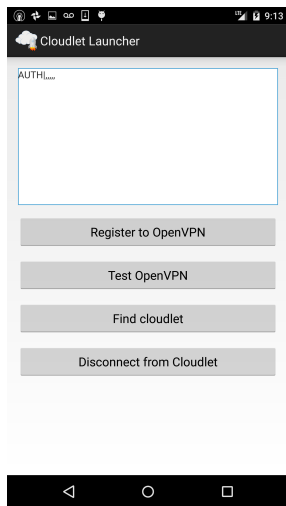


Figure 2.7: Cloudlet Launcher UI

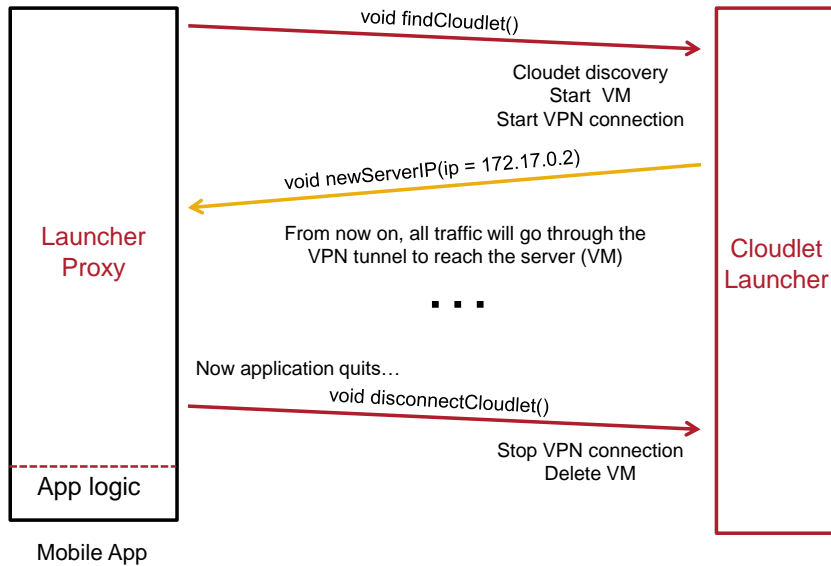


Figure 2.8: Example API Calls between Cloudlet Launcher and Launcher Proxy

Cloudlet Launcher Activity

While Cloudlet Launcher is run mainly as the background service, I have also created an Android activity associated with it that offers a simple user interface. This interface allows a user to configure cloudlet service related settings, such as user ID and VPN keys, as well as to test and debug Cloudlet Launcher service. Figure 2.7 shows an example graphical user interface in this activity.

Transforming Existing Applications

While the design of Cloudlet Launcher intends to minimize the effort of an existing application to leverage a cloudlet, each application (e.g. Gabriel front-end software) still needs to implement some glue logic, which we call the launcher proxy, to leverage Cloudlet Launcher service. The launcher proxy is in charge of communicating to the Cloudlet Launcher service, such as connecting to the service (e.g. in `onStart()`), disconnect from the service (e.g. in `onDestroy()`), implementing callbacks such as when a new VM has been provisioned, etc. This glue logic should be simple enough to be implemented for any new applications. Figure 2.8 shows example function call sequences between a launcher proxy and Cloudlet Launcher.

2.4 Micro Benchmark Evaluation

In this section, I will use several micro benchmarks to evaluate Gabriel’s performance. These are not full-fledged cognitive assistance applications, but rather some representative components that

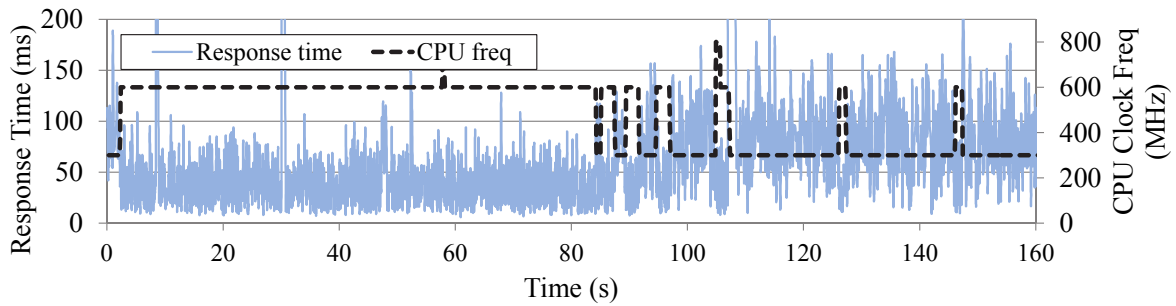


Figure 2.9: Trace of CPU Frequency, Response Time Changes on Google Glass

test the system from different aspects, including its VM overhead, the effectiveness of end-to-end flow control, and its ability to scale out through the pub-sub backbone. Thorough evaluation with realistic applications will be presented in Chapter 3.

2.4.1 Experimental Setup

I have set up a desktop-class machine running OpenStack [136] to represent a cloudlet. This machine has an Intel® Core™ i7-3770 quad-core CPU that runs at 3.4GHz, and 15GB of memory. Since this section focuses on studying the Gabriel back-end structure, I use Google Glass as the client device in all experiments. Chapter 3 will evaluate the system performance with various mobile devices.

The Google Glass always connects to the cloudlet via a dedicated Wi-Fi Access Point (AP) which routes over the public university network. I use a pre-recorded 360p video to be fed to the system. During experiments, the Glass device captures frames with camera on, but replaces them with frames from the pre-recorded video. This ensures that the measurements are consistent and reproducible. Each experiment typically runs for five minutes.

While measuring latency, I noticed anomalous variance during experiments. I found that to reduce discomfort as a wearable device, Google Glass attempts to minimize heat generation by scaling CPU frequency. Unfortunately, this causes latency to vary wildly as seen in Figure 2.9. The CPU operates at 300, 600, 800, or 1008 MHz, but uses the higher frequencies only for short bursts of time. Empirically, with ambient room temperature around 70 degrees Fahrenheit, Glass sustains 600 MHz for a few minutes at a time, after which it drops to its lowest setting of 300 MHz. In Figure 2.9, I let Glass capture images from its camera, send them over Wi-Fi to a server, and report the time from capture to the time of server acknowledgment back to the Glass. Whenever the CPU frequency drops, there is a significant increase in latency. To reduce thermal variability as a source of latency variability in my experiments, I externally cool the Glass device by wrapping it with an ice pack in every experiment, allowing it to sustain 600 MHz indefinitely.

percentile	1%	10%	50%	90%	99%
delay (ms)	1.8	2.3	3.4	5.1	6.4

The delay between receiving a sensor data and sending a result using *NULL* module.

Table 2.2: Intrinsic Delay Introduced by Gabriel

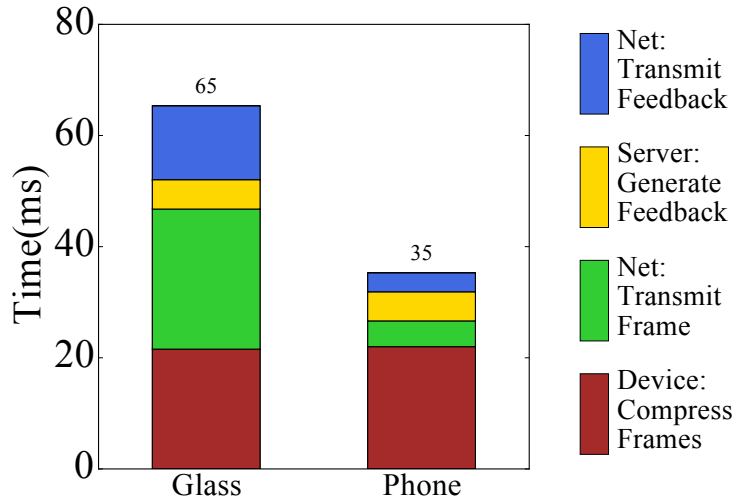


Figure 2.10: Time Breakdown of Mean End-to-end Latency for the *NULL* Cognitive Module

2.4.2 Gabriel Overhead

Gabriel uses VMs extensively so that there are few constraints on the operating systems and cognitive modules used. However, VMs are known to have an overhead on system processing speed. To quantify the overhead of Gabriel, I measure the time delay on a cloudlet between receiving sensor data from a Glass device and sending back a result. To isolate performance of the infrastructure, I use a *NULL* cognitive module which simply sends a dummy result upon receiving any data. The Cognitive VM is run on a separate physical machine from the Control and User Guidance VMs. Table 2.2 summarizes the time spent in Gabriel backend for 3000 request-response pairs as measured on the physical machine hosting the Control VM. This represents the intrinsic overhead of Gabriel. At 2–5 ms, it is surprisingly small, considering that it includes the overheads of the Control, Cognitive, and User Guidance VM stacks, as well as the overhead of traversing the cloudlet’s internal Ethernet network.

Figure 2.10 shows the end-to-end system response time of Gabriel for the *NULL* application. For both devices used, the server processing time (overhead of Gabriel backend) is really small, confirming the results in Table 2.2. When using a modern smartphone, the major portion of time is spent on encoding video stream on the mobile device. Using Google Glass significantly increases the networking time for both uplink and downlink, because of the older version of WiFi technology being used. Section 3.3.6 will explain in detail the performance difference between devices. In general, Gabriel platform achieves 65 and 35 ms mean latency for Google Glass and phone, respectively, suggesting that I have met the goal of tens of milliseconds as set

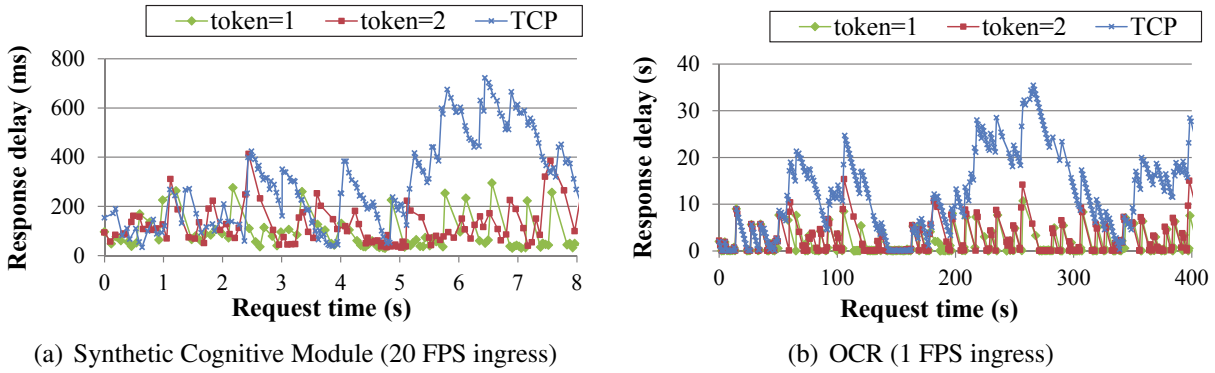


Figure 2.11: Response Delay of Request

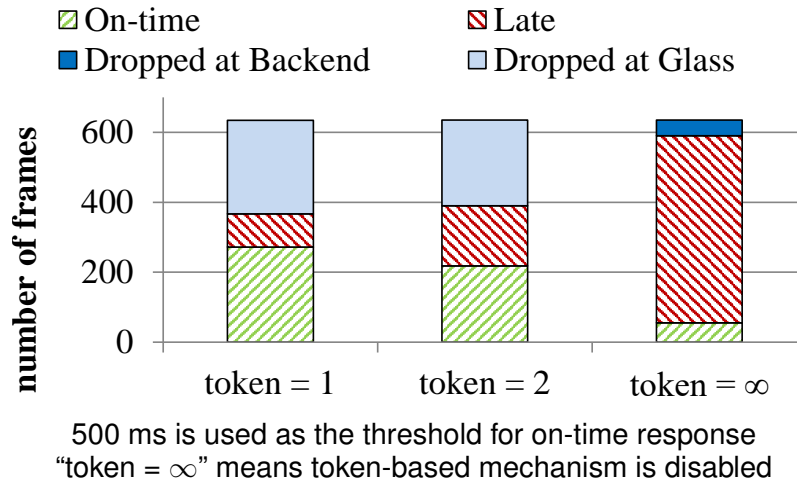


Figure 2.12: Breakdown of Processed and Dropped Frame for OCR

in Section 2.1.1. These end-to-end latencies represent the minimum achievable for offload of a trivial task. Real cognitive computations will add to these values.

2.4.3 Queuing Delay Mitigation

In order to evaluate the need for the application-level, end-to-end flow-control mechanism for mitigating queuing, I first consider a baseline case where no explicit flow control is done. Instead, I push data in an open-loop manner, and rely on average processing throughput to be faster than the ingress data rate to keep queuing in check. Unfortunately, the processing times of cognitive modules are highly variable and content-dependent. An image that takes much longer than average to process will cause queuing and an increase in latency for subsequent ones. However, with my two-level token bucket mechanism, explicit feedback controls the input rate to avoid building up queues.

I first compare the differences in queuing behavior with and without the flow control mechanism using a synthetic cognitive module. To reflect high variability, the synthetic application

has a bimodal processing time distribution: 90% of images take 20 ms, 10% take 200 ms each. Images are assigned deterministically to the slow or fast category using a content-based hash function. With this distribution, the average processing time is 38 ms, and with an ingress interval of 50 ms (20 fps), the system is underloaded on average. However, as shown in Figure 2.11(a), without application-level flow control (labeled TCP), many frames exhibit much higher latencies than expected due to queuing. In contrast, the traces using my token-based flow control mechanism with either 1 or 2 tokens reduce queuing and control latency. There are still spikes due to the arrival of slow frames, but these do not result in increased latency for subsequent frames. I repeat the experiment with a real OCR module. Here, the mean processing time is just under 1 second, so I use an offered load of 1 FPS. The traces in Figure 2.11(b) show similar results: latency can increase significantly when explicit flow control is not used (TCP), but remains under control with the token-based flow control mechanism.

The tight control on latency comes at the cost of throughput – the token based mechanism drops frames at the source to limit the number of data items in flight through the system. Figure 2.12 shows the number of frames that are dropped, processed on time, and processed but late for the OCR experiment. Here, I use a threshold of 0.5 second as the “on-time” latency limit. The token-based mechanism (with tokens set to 1) drops a substantial number of frames at the source, but the majority of processed frames exhibit reasonable latency. Increasing tokens (and number of frames in flight) to 2 reduces the dropped frames slightly, but results in more late responses. Finally, the baseline case without the flow-control mechanism attempts to process all frames, but the vast majority are processed late. The only dropped frames are due to the output queues filling and blocking the application, after the costs of transmitting the frames from the Glass front-end have been incurred. In contrast, the token scheme drops frames in the Glass device before they are transmitted over Wi-Fi. These results show that my token-based flow control mechanism is effective and necessary for reducing system latency.

2.4.4 System Performance with Multiple Cognitive VMs

All previous experimental results examine the performance of Gabriel with one Cognitive VM. This section stresses the pub-sub backbone of Gabriel, and tests its ability to scale out multiple machines to run a very complex application. To do this, I have developed seven cognitive modules, many of which are based on available research and commercial software. The rest of the section will first detail each cognitive module, and then report the performance of the entire system when all cognitive VMs are running together. The different modules are summarized in Figure 2.13.

Supported Cognitive Modules

Face Recognition: A most basic cognitive task is the recognition of human faces. The face recognition module runs on Windows. It uses a Haar Cascade of classifiers to perform detection, and then uses the Eigenfaces method [171] based on principal component analysis (PCA) to make an identification from a database of known faces. The implementation is based

Module	Source	OS	VCPU*
Face Recognition	open source research code based on OpenCV [171]	Windows	2
Object (MOPED)	based on published MOPED [121] code	Linux (32-bit)	4
Object (STF)	re-implementation of STF [156] algorithm	Linux	4
OCR (Open Source)	open source Tesseract-OCR [67, 160] package	Linux	2
OCR (Commercial)	closed-source VeryPDF OCR product [174]	Windows	2
Motion Classifier	action detection based on research MoSIFT [41] code	Linux	12
Augmented Reality	research code from [165]	Windows	2

* Number of virtual CPUs allocated to the Cognitive VM at experiment

Figure 2.13: Summary of Implemented Cognitive Modules for Evaluation

on OpenCV [8] image processing and computer vision routines.

Object Recognition (MOPED): The evaluation runs two different object recognition modules. They are based on different computer vision algorithms, with different performance and accuracy characteristics. The open source MOPED module runs on Linux, and makes use of multiple cores. It extracts key visual elements (SIFT features [119]) from an image, matches them against a database, and finally performs geometric computations to determine the pose of the identified object. The database is populated with thousands of features extracted from more than 500 images of 13 different objects.

Object Recognition (STF): The other object recognition module is based on machine learning, using the semantic texton forest (STF) algorithm described by Shotton et al [156]. The MSRC21 image dataset mentioned in that work (with 21 classes of common objects) was used as the training dataset. The Python-based implementation runs on Linux and is single-threaded.

OCR (Open Source): A critical assistance need for users with visual impairments is a way to determine what is written on signs in the environment. Optical character recognition (OCR) on video frames captured by the Glass camera is one way to accomplish this. My evaluation is based on two different OCR modules. One of them is the open source Tesseract-OCR package [67, 160], running on Linux.

OCR (Commercial): The second supported OCR module is a Windows-based commercial product: VeryPDF PDF to Text OCR Converter [174]. It provides a command-line interface that allows it to be scripted and readily connected to the rest of the Gabriel system.

Motion Classifier: To interpret motion in the surroundings (such as someone is waving to a user or running towards him), I have implemented a cognitive module based on the MoSIFT algorithm [41]. From pairs of consecutive frames of video, it extracts features that incorporate aspects of appearance and movement. These are clustered to produce histograms that character-

Module	FPS	Response time (ms)			CPU load(%)
		10%	50%	90%	
Face	4.4	389	659	929	93.4
Object (MOPED)	1.6	962	1207	1647	50.9
Object (STF)	0.4	4371	4609	5055	24.9
OCR (Open)	14.4	41	87	147	24.1
OCR (Comm)	2.3	435	522	653	95.8
Motion	14	152	199	260	20.1
AR	14.1	72	126	192	83.8

Table 2.3: FPS and Latency of Cognitive Modules

ize the scene. Classifying the results across a small window of frames, the module detects if the video fragment contains one of a small number of previously-trained motions, including waving, clapping, squatting, and turning around.

Augmented Reality: The last module is a Windows-based augmented reality module that identifies buildings and landmarks in images [165]. It extracts a set of feature descriptors from the image, and matches them to a database that is populated from 1005 labeled images of 200 buildings. The implementation is multi-threaded, and makes significant use of OpenCV libraries and Intel Performance Primitives (IPP) libraries. Labeled images can be displayed on the Glass device, or their labels can be read to the user.

System Performance

In this experiment, I used a different pre-recorded video which includes footage recognizable by the cognitive modules such as signs with text for OCR, a Coke can for object recognition, human faces for face recognition, and waving for motion recognition. Figure 2.3 shows the performance of each offloading module during this full system benchmark. Frame per second (FPS) and response time is measured at the Glass device. The results reflect that the cognitive modules operate independently within their VMs. Most importantly, the overall system is not limited to the slowest cognitive module. The cognitive modules that can process frames quickly operate at higher frame rates than the slower cognitive modules, which is precisely what Gabriel promises. In addition, the quickest cognitive modules maintain end-to-end latency of tens of milliseconds.

Since the group of Cognitive VMs are dedicated to a single user without being shared, the number of total VMs is proportional to the number of users. Therefore, it is important to assign appropriate amounts of hardware resources to each cognitive module. In the experiment, the CPU usage of each cognitive module ranges from 20.1% to 95.8% as in Figure 2.3. Many of the low CPU utilizations are due to limited parallel sections in the cognitive module implementations, thus under-utilizing some of the cores. For example, MOPED uses four cores in some sections, but only one or two for the remainder, for an average of 50% utilization. This implies one may be able to statistically multiplex multiple cognitive modules on the same physical cores, but this will have some impact on latency.

One should also consider the overhead of provisioning a cloudlet for the first time. Unlike the cloud, where all necessary VM images can be assumed to be present, a new cloudlet may need to be provisioned dynamically. VM synthesis [74] can be used to rapidly provision a cloudlet from the cloud. With this technique, it takes around 10 seconds to provision and launch the control and User Guidance VMs from Amazon EC2 Oregon to the cloudlet on the CMU campus. The provisioning time for a Cognitive VM ranges from 10 seconds to 33 seconds depending on VM size. In other words, just 10 seconds after associating with a new cloudlet, a user can start transmitting sensor data to the Gabriel back-end. It is important to note that provisioning is a one-time operation. Persistent caching of VM images on a cloudlet ensures that re-provisioning is rarely needed.

Chapter 3

Wearable Cognitive Assistance Applications

On top of the Gabriel platform introduced in the previous chapter, I have built realistic cognitive assistance applications for the nine tasks summarized in Table 3.1. These applications cover a variety of usage scenarios, and leverage a broad set of computer vision and signal processing techniques. Thus, they form a benchmark suite that can be used to thoroughly evaluate the performance of different cognitive assistance systems. In particular, I will use them to evaluate the performance of Gabriel under different system configurations. Throughout this thesis, I use the capitalized application name (the first column in Table 3.1) to denote the specific assistive application I have built (e.g. Pool for Pool Assistant). YouTube video demos of some applications are available at <http://goo.gl/02m0nL>.

This chapter is organized as follows. First, Section 3.1.1 will give an overview of the diverse applications, as well as their structural similarity that makes Gabriel a suitable underlying platform. The rest of Section 3.1 will then detail the implementation of each application. Next, Section 3.2 derives a target latency for each of the applications using a variety of approaches, including user studies. Finally, in Section 3.3, I will study how the performance of such applications can be affected by different system configurations, such as mobile and server hardware, the proximity of offloading sites, and networking technology. These measurements will focus on end-to-end latency, since this is the metric that directly reflects the application quality a user experiences. Measurements regarding system throughput and energy consumption will be discussed in Chapter 4 and Chapter 5, respectively.

3.1 Application Implementation

3.1.1 Overview: Application Diversity and Similarity

The applications span a wide variety of guidance tasks. All use computer vision (CV), but with substantially different algorithms, ranging from color and edge detection, face detection and



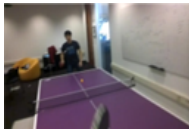



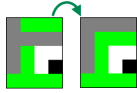

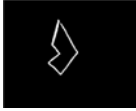







App Name	Example Input Video Frame	Description	Symbolic Representation	Example Guidance
Pool		Helps a novice pool player aim correctly. Gives continuous visual feedback (left or right arrow, or thumbs up) as the user turns the cue stick. Uses color, line, and shape detection.	<Pocket, object ball, cue ball, cue top, cue bottom>	
Ping-pong		Tells novice to hit ball to the left or right for better chance of winning a rally. Uses motion detection to track opponent. Whispers “left” or “right” or offers spatial audio guidance using [166].	<InRally, ball position, opponent position>	Whispers “Left!”
Work-out		Guides correct user form in exercise actions like sit-ups and push-ups, and counts out repetitions. Uses Volumetric Template Matching [103] to classify the exercise. Use third-person viewpoint.	<Action, count>	Says “8 ”
Face		Whispers the name of a person whose name you cannot recall. Applies state-of-art face recognizer using deep residual network [80].	ASCII text of name	Whispers “Barack Obama”
Lego		Guides a user in assembling 2D Lego models. Extracts a matrix to represent Lego shape and color being assembled. Provides both visual animation guidance and auditory instructions.	[[0, 2, 1, 1], [0, 2, 1, 6], [2, 2, 2, 2]]	 “Put a 1x3 green piece on top”
Draw		Helps a user to sketch better, using any drawing surface and instrument. Compares a user’s sketch with masters’ drawing in the database, and displays the error alignment.		
Sandwich		Helps a cooking novice prepare sandwiches according to a recipe. Experimented with plastic ingredients. Object detection follows the state-of-art faster-RCNN deep neural net approach [146].	Object: “E.g. Lettuce on top of ham and bread”	 Says “Put bread on the lettuce”
Furniture		Helps a user to assemble Ikea furniture (e.g. lamp/stool). Uses similar object detection techniques as in Sandwich. Provides a short video tutorial in response to the user’s progress for next-step instruction.	Name and position of objects: E.g. Shade + pipe + base.	 Says “Last step! Install the bulb.”
Graffiti		Allows users to annotate a real world object or scene with sketches. Other users can see these annotations as they point their phone to the same object/scene. Uses SIFT features [119] for image matching and color histograms to filter out false positives.	Detected SIFT feature vectors and computed color histograms	

Table 3.1: Prototype Wearable Cognitive Assistance Applications Developed and Studied

recognition, to object recognition based on deep neural networks (DNNs). Some, such as Lego, Sandwich, and Furniture, provide step-by-step guidance to completing a task. They determine whether the user has correctly completed the current step of the task, and then provide guidance for the next step, or indicate corrective actions to be taken. In this regard, they resemble GPS-navigation, where step-by-step instructions are provided, and corrective actions are suggested if the user makes a mistake or deviates from the plan. Other applications are even more tightly interactive. For example, Ping-pong provides guidance on whether the user should hit left or right based on the opponent position and ball trajectory during a rally. Pool provides continuous guidance as the player aims the cue stick. For both, the feedback needs to be immediate, and thus requires very low end-to-end latency for the entire cognitive assistance processing stack.

In spite of these differences, there are significant commonalities between the applications. First, they are all based on visual understanding of scenes, and rely on video data acquisition from a mobile device. For all of my applications except Workout, this is a first-person view from a wearable device.

Additionally, all of these applications utilize a similar 2-phase operational structure. In the first phase, the sensor inputs are analyzed to extract a *symbolic representation* of task progress. This is an idealized representation of the input sensor values that contains all of the useful information for a certain task, and excludes all irrelevant detail. This phase has to be tolerant of considerable real-world variability. For example, it has to be tolerant of variable lighting levels, varying light sources, varying positions of the viewer with respect to the task artifacts, task-unrelated clutter in the background, and so on. One can view the extraction of a symbolic representation as a task-specific “analog-to-digital” conversion: the enormous state space of sensor values is simplified to a much smaller state task-specific space. In the Lego task, for example, the symbolic representation is a two-dimensional matrix. Each matrix element is a small integer that represents the color of the corresponding Lego brick, with zero indicating the absence of a brick.

The second phase of each task workflow operates solely on the symbolic representation. It requires task specific domain knowledge to generate appropriate guidance. For example, in Pool, the application has to calculate correct shot angle based on the position of the balls and the pocket. In some applications, the guidance generation is not solely based on the current symbolic representation, but has to leverage previous user states as well. Therefore, it is also the job of this phase to keep a history of symbolic representations for the recent past.

The common two-phase processing pipeline makes Gabriel a perfect underlying platform to implement all these applications. The first phase is usually based on complex computer vision components such as object detection and face recognition, which fit nicely in Cognitive VMs. The second phase is very application specific, and can be implemented in the User Guidance VM. The inputs of the User Guidance VM are therefore the symbolic representations of different applications. In summary, this suite of applications covers a broad, representative range of the emerging class of interactive cognitive assistance applications. As these applications support multiple client devices, and both cloud and cloudlet backends, they can help us examine how various aspects of the system affect performance.

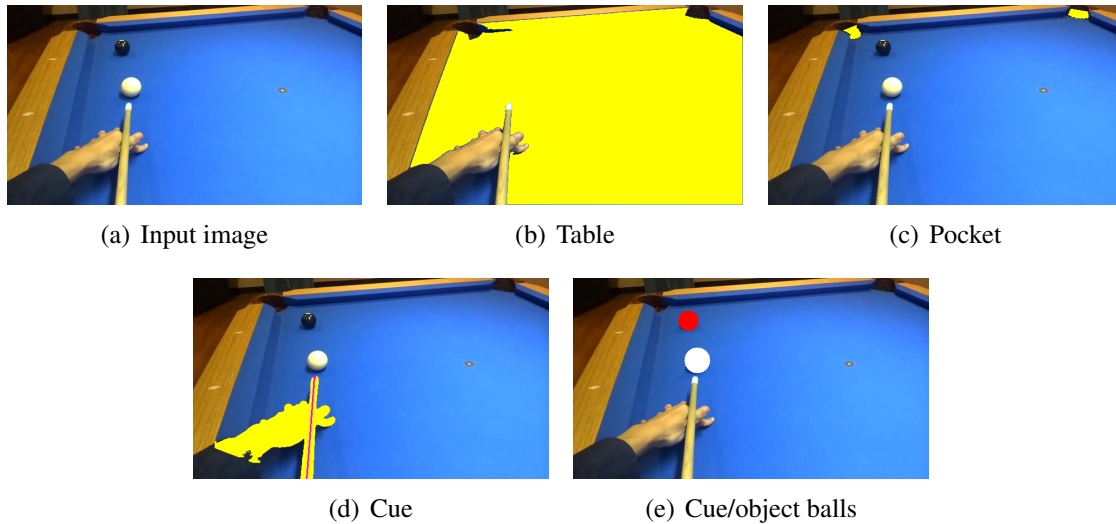


Figure 3.1: Pool: Extracting Symbolic Representation

3.1.2 Pool Assistant

The first assistive application, Pool, helps a novice pool player to find the correct angle to aim in playing a billiard game. It gives the user continuous visual feedback (left arrow, right arrow, or thumbs up) as the user turns her cue stick when aiming. This smart-glass-based application is fully portable, in contrast to other pool aiming applications, such as ARPool [2], that require a fixed camera to capture images, and a projector to offer the user feedback.

As shown in Table 3.1, the symbolic representation of Pool is a five tuple that describes the positions of the cue ball, object ball, target pocket, and the top and bottom of the cue stick. The application is mainly implemented with OpenCV [8] image processing library to extract this representation. Using the video frame shown in Figure 3.1(a) as a working example, Figure 3.1 illustrates the application pipeline step by step. The area of the table is first detected by setting bounds on the HSV (hue, saturation, and value) color space of the image. Loose bounds are applied to detect a rough area and to understand the current lighting condition. Based on this, a tight bound is set to more accurately detect the table contour (Figure 3.1(b)). The hands, cue stick, and pockets are detected by using the fact that they are the only things that make the table contour non-convex. The concave parts far from the user (upper part of image) are deemed to be the pockets (Figure 3.1(c)), while those close to the user (lower part of image) are deemed to be the hands and the cue stick. Line detection is then used to accurately locate the cue stick (Figure 3.1(d)). Balls are detected as “holes” inside the table using Hough Circle Transform [53] (Figure 3.1(e)). The cue ball and object ball are distinguished by their differing distances to the cue stick.

The positions in the symbolic representation are pixel values in the image. These values, in combination with an estimation of users viewing height and position, give an estimation of the ideal shot angle, based on the widely used fractional aiming system [6]. The deviation of the cue stick from this angle is used as the basis of feedback. For example, if the current aiming direction

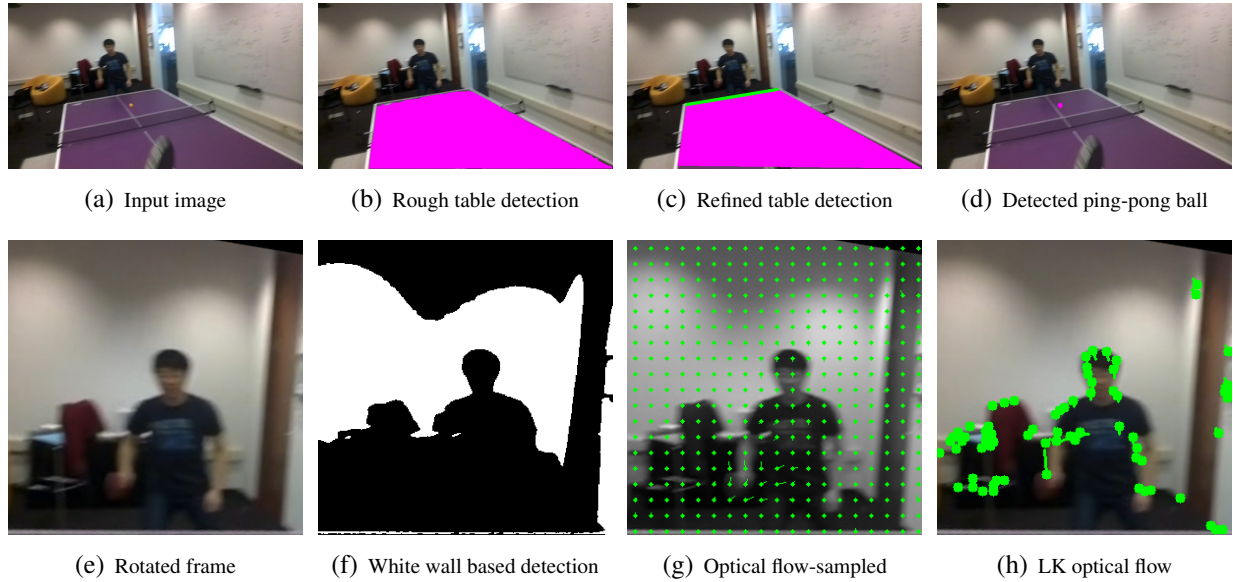


Figure 3.2: Ping-pong: Symbolic Representation

is far to the left, a right arrow will be shown.

3.1.3 Ping-pong Assistant

Another application that has very tight real-time constraints is Ping-pong. A YouTube demo shows how it works: https://youtu.be/_lp32sowyUA. Its goal is to help a user to choose the optimal direction to hit the ball to the opponent, so that she has a better chance to win the rally. Note that this is not intended for training a professional player. Its purpose is merely to help a novice play a little better by whispering hints.

Ping-pong uses a 3-tuple as the symbolic representation. The first element is a boolean indicating whether the player is in a rally. The second element is a floating point number in the range from 0 to 1, describing the position of the opponent (extreme left is 0 and extreme right is 1). The third element uses the same notation to describe the position of the ball.

Table detector: As shown in Figure 3.2(b), simple color detection can be used to approximately locate the table. To accurately detect the edges of the table in spite of lighting variation and occlusions by the opponent, I dilate the detected area, use white table edges to remove low possibility areas, and then use the Douglas-Peucker algorithm [52] to approximate the table area with polygons. Multiple iterations of this procedure yield a clean final result (Figure 3.2(c)). The top edge is also detected using the polygon information and marked in green in the figure.

Ball detector: For every area whose color is close to yellow, I determine if it is the ball based on its shape, size, position relative to table, and the ball's position in the previous frame. Once the ball is detected (Figure 3.2(d)), I find its coordinates in the input image and calculate its position

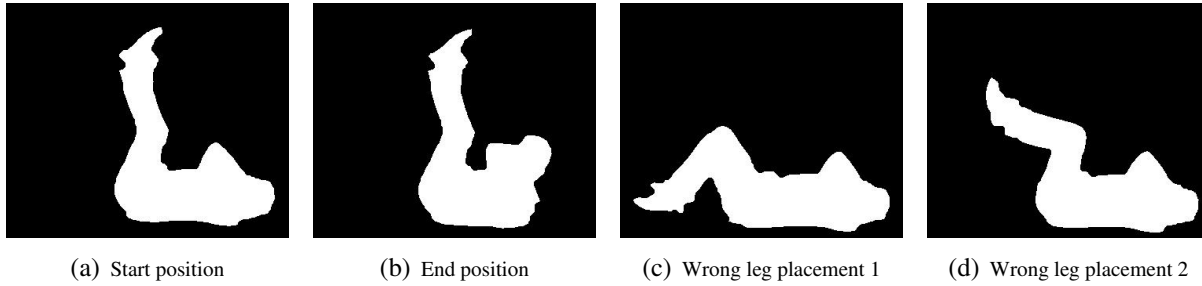


Figure 3.3: Workout: Example Volumetric Template for Sit-ups

relative to the table by using the transformation matrix described below for the opponent detector.

Opponent detector: Using the information obtained in table detection, I rotate and cut the area above the top edge of the table to get a square shape containing the opponent (Figure 3.2(e)). Within this standardized area, I use three approaches to locate the opponent. The first approach is simple and fast, but error-prone. It assumes that the background is mostly white wall, and selects the area with least white color (Figure 3.2(f)). The second approach is based on human motion calculated by optical flow [86], using two consecutive frames as input. It samples the image and calculates the flow at each sample point. The small green dots in Figure 3.2(g) are the sample points, while the arrows describe the direction and magnitude of flow. By appropriate low-pass filtering, I can find the position with strongest motion, and hence the opponent’s likely location. The third approach is similar to the second, but calculates flow only at detected interest points (green circles in Figure 3.2(h)). The results from these three approaches are combined using a weighted average for optimal accuracy.

Guidance is based on a recent history of states. For example, if the application knows the opponent is standing at the right, and the ball has been hit to the right several times, it will suggest that the user hit to the left. Currently, Ping-pong uses simple auditory guidance only, just “left” or “right.” When appropriate hardware is available, this could be replaced by spatial audio guidance using [166].

3.1.4 Workout Assistant

I have created a Workout Assistant that serves as a virtual coach for exercises such as sit-ups and push-ups. It continuously monitors the actions a user is performing, checks her form, and counts out repetitions. In contrast to using on-body sensors to count repetitions, my vision-based implementation has the potential to also offer guidance on the correctness of the user’s form.

While all other applications I have implemented are based on first-person vision, Workout uses a smartphone placed on the floor to capture the user doing the action. This is because the first-person viewpoint provided by smart glasses is unsuitable for capturing a video segment of the users exercise: a third-person viewpoint is necessary. It may be possible to use a mirror to provide a third-person viewpoint for a smart glass user, but I have not explored such an imple-

mentation.

As shown in Table 3.1, the symbolic representation of Workout has two parts: an integer that encodes the exercise being performed (e.g. pushup), and another integer indicating the repetition count. This information is obtained by analyzing a video segment that is typically 10-15 frames in length. Volumetric Template Matching [103] is applied to such group of frames to compare a user’s form with that of a professional athlete, which then determines the exercise being performed. For example, Figure 3.3(a) and 3.3(b) show the starting and ending frames of the template for a sit-up with legs straight up. These templates are created beforehand by carefully removing the background pixels of a video capturing the athlete’s performance. To speed up background subtraction, I recorded the video with relatively clean background, and then applied techniques such as [104] for automatic background subtraction.

If a user’s action is a good match to the known template, the count is incremented and then spoken aloud. For example, to be successfully recognized by the application, a user needs to always have legs straight up when performing sit-ups. Workout’s template matching classifies a poorly-performed repetition as a distinct type of exercise (e.g. if a user’s action matches Figure 3.3(c) or 3.3(d), it will be classified as “bad situp”).

3.1.5 Face Assistant

Everyone has experienced situations where you see a familiar face but cannot remember the name of the person, or the context of your previous encounters with that person. To help in such situations, I have built a Face Assistant application. Currently it only whispers the person’s name, but the application can be easily extended to provide more contextual information from a private or public database. For example, it could be used in combination with Expression [25] to offer conversational hints.

On each video frame, the Gabriel back-end detects faces and extracts a tightly-cropped image of each face. It then presents the image to the state-of-the-art face recognizer using deep residual network [80] implemented using Dlib [105] version 19.4. The DNN model is similar to the previous work of DeepFace [164] and FaceNet [152], and is trained upon millions of face images. The result of the model is a 128 dimensional vector that represents an face in an Euclidean space. This vector is then passed to a SVM classifier that tells the identity of each face. The incremental training to recognize a small number of new faces only involves retraining the SVM classifier while keeping the DNN model unchanged, and can be done within a few tens of seconds on the cloudlet.

As indicated in Table 3.1, the symbolic representation is the name (ascii text) of the person corresponding to the face. The task guidance is the spoken name (audio). In a more sophisticated implementation, the symbolic representation could be the primary key of the database entry corresponding to the person, and the guidance could be the name and additional information about the person from the database.

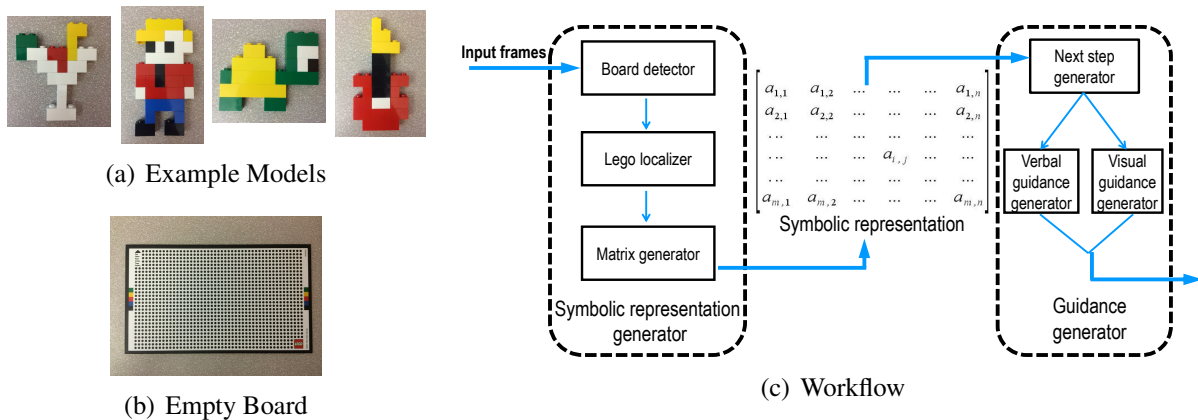


Figure 3.4: Lego Task

3.1.6 Lego Assistant

My Lego Assistant guides a user in assembling 2D models using the Lego product *Life of George* [110]. It not only provides step-by-step instructions, like in the original instruction sheets, on how to assemble a complex shape, but also checks a user’s progress and mistakes in real time. Figures 3.4(a) and 3.4(b) show some example Lego models and the board on which they are assembled. A YouTube demo of this application is at <http://youtu.be/uy17Hz5xvmY>.

To extract the symbolic representation of the example image in Figure 3.5(a), the first step is to find the board, using its distinctive black border and black dot pattern. Reliable detection of the board is harder than it may seem because of variation in lighting conditions. I subtract the original image by a blurred version of it, and then threshold the difference to give a robust black detector (Figure 3.5(b)). I find the boundary of the board (Figure 3.5(c)), then the four corners (Figure 3.5(d)), and then perform perspective transformation (Figure 3.5(e)).

Next, I extract the Lego model from the board. I perform edge detection (Figure 3.5(f)), then apply dilations and erosions to find the largest blob near the center of the board (Figure 3.5(g)). Unfortunately, sole reliance on edge detection is not robust. The sides of Lego bricks have more texture than the surface, leading to uncertainty in detection. I correct for this by finding the sides using color detection, adding them to the Lego shape obtained by edge detection (Figure 3.5(h)), and then performing erosion to remove the side parts (Figure 3.5(i)). The amount of erosion is calculated from the perspective transform matrix. For robust color detection, I use the grey world color normalization method [37].

In the final set of steps, I rotate the image to an upright orientation (Figure 3.5(j)). Each pixel is then quantized to one of the Lego brick colors (Figures 3.5(k) and 3.5(l)), with magenta color representing uncertainty. Final assignment of brick color is done by a weighted majority vote of colors within the block, with pixels near the block center being assigned more weight. Figure 3.5(m) shows the final matrix representation. Figure 3.5(n) is synthesized from this matrix.

A task in this application is represented as a linked list of Lego states, starting from the

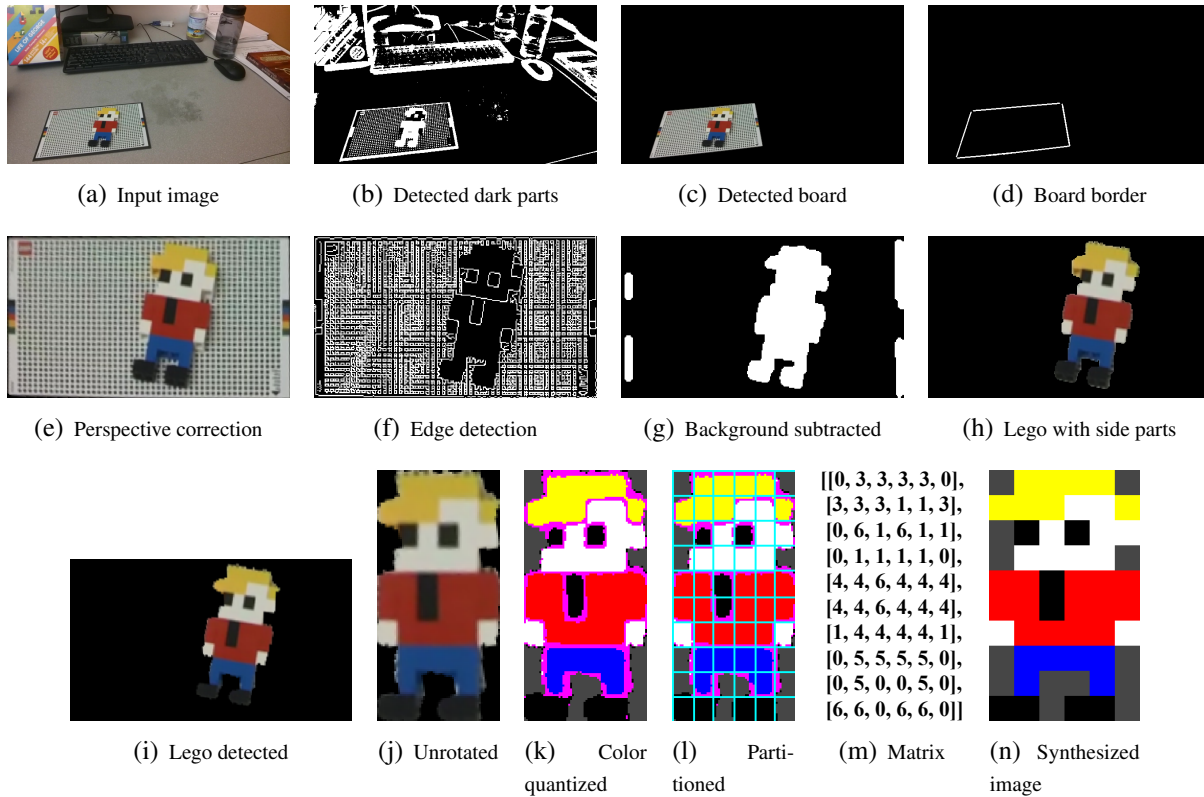


Figure 3.5: Lego: Symbolic Representation

beginning state (nothing) to the target state (the user's goal). Figure 3.6 shows an example of a task, with each state being illustrated by the synthesized image of the matrix that represents that state. Based on the matrix representation of the current Lego state, Lego tries to find an optimal next step towards the task target. In practice, my implementation consists of the four logical steps below:

1. I check if any state in the task representation (linked list) is strictly one brick more than the current user state. If so, I ask the user to add the brick to the existing Lego model; otherwise go to step 2.
2. I check if the current user state can match any state in the task representation (linked list) by moving an existing brick. If so, I ask the user to move the brick, otherwise go to step 3.
3. I check if any state in the task representation (linked list) is strictly one brick less than the current user state. If so I ask the user to remove the brick; otherwise go to step 4.
4. At this step, the system does not know exactly what brick to ask the user to add, move, or remove, which means the user has done something quite different from the instructions. I will then ask the user to revert back to a state that she has built earlier, and go through the steps 1-3 again.

Generally, the instruction to the user consists of three parts, the action the user needs to

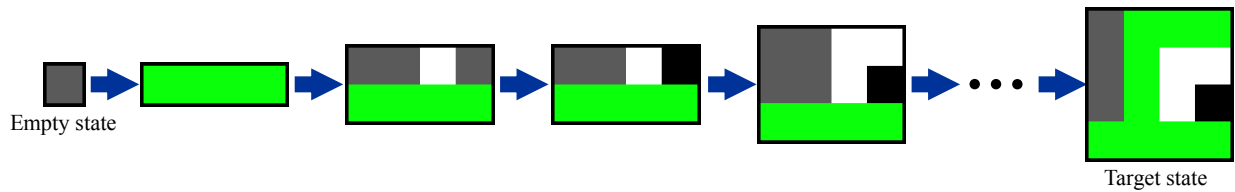


Figure 3.6: Example State List Representation of Lego Task

perform (add, move, or remove), the location information about the action (e.g. where to add the brick or which one to move), and the description of the brick (e.g. size and color). I use both auditory and visual guidance for Lego. The auditory text guidance is a complete instruction whispered into the user’s ears through text-to-speech technology. The visual guidance is an animation displayed on the wearable device’s screen, showing how the bricks should be placed (as shown in Table 3.1).

3.1.7 Drawing Assistant

In contrast to many other applications, which are entirely new, the Drawing Assistant modifies an existing application to provide wearable cognitive assistance. The *Drawing Assistant* by Iarussi et al. [93] guides a user in the classic technique of drawing-by-observation. As originally implemented, the application requires use of a special input drawing device, such as a pen-tablet. On the computer screen, it offers construction lines for the user to follow, recognizes the user’s drawing progress, and offers corrective feedback. For example, Figure 3.8(a) shows a target drawing that the user is trying to copy. The blue polygon in Figure 3.8(b) is the outline generated by Draw, and Figure 3.8(c) shows the feedback provided by the application in response to the user’s attempt to copy the polygon. The color of the construction lines in feedback images represents the correctness of different parts of the user’s attempt: blue is good, red is bad. The dashed red lines indicate erroneous alignment between corners.

This application, as well as many other similar ones (e.g. [109]), requires the use of computer-friendly input (pen-tablet or mouse) and output (screen). Draw uses smart glasses to extend this application to work with arbitrary drawing instruments and surfaces: e.g., using pencil on paper, oil paint on canvas, or colored markers on a whiteboard. It uses cloudlet processing to extract a symbolic representation of the user’s input. I then splice this extracted information into the existing guidance logic of Draw, and the rest of the system works with little change. A YouTube demo of this application is at <https://youtu.be/nuQpPtVJC6o>.

For ease of exposition, I assume that the user is drawing with a pen on paper. The processing for other media (e.g. colored markers on a whiteboard) is very similar. To understand a user’s drawing, the first step is to locate the paper from any arbitrary background. Similar to the black detection problem in Lego, detecting “white” paper is not as easy as looking directly at the RGB values. Therefore, I use an approach similar to the board localization approach in Lego to reliably detect the easel surface (Figure 3.7(b)). This is followed by a quadrilateral detection which represents the paper (Figure 3.7(c)). Line and corner detection are then applied to transform the paper image to a standard rectangle shape (Figure 3.7(d)).

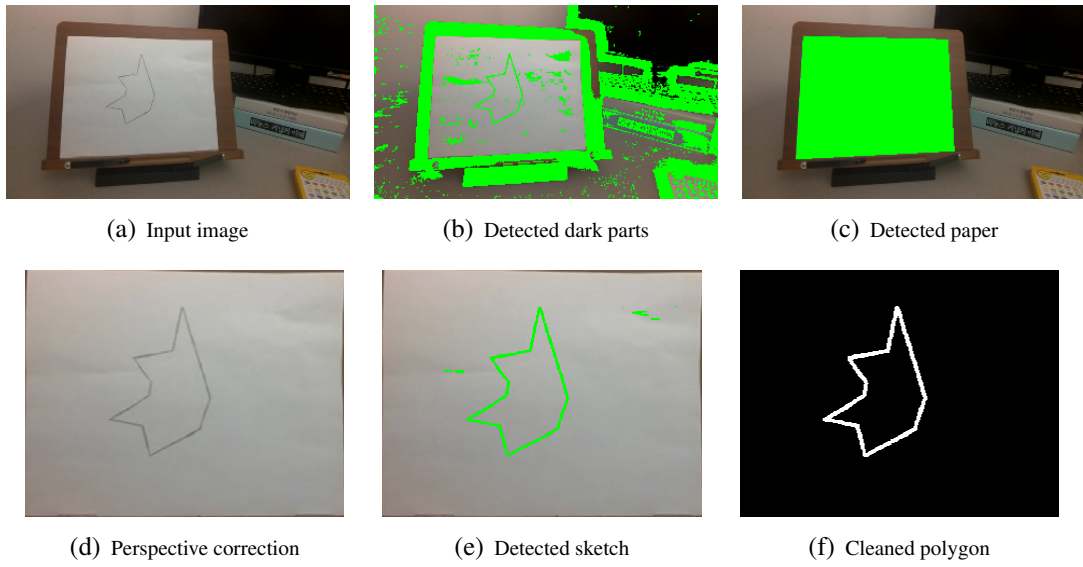


Figure 3.7: Draw: Symbolic Representation

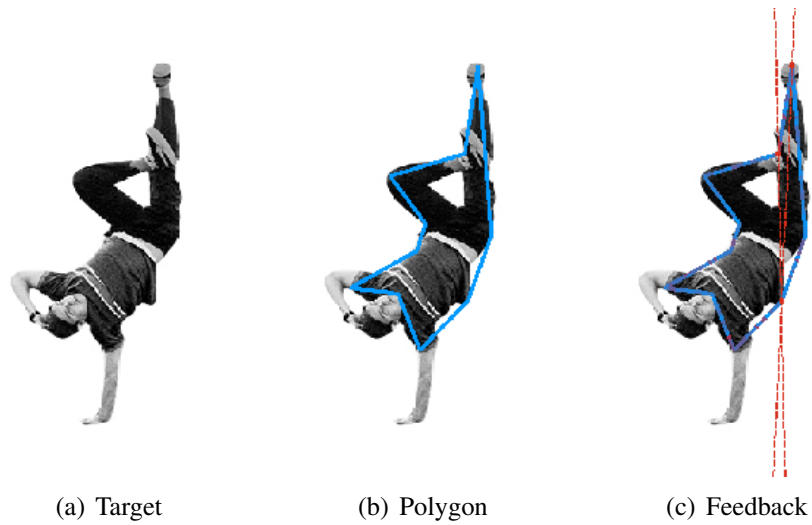


Figure 3.8: Guidance by Drawing Assistant

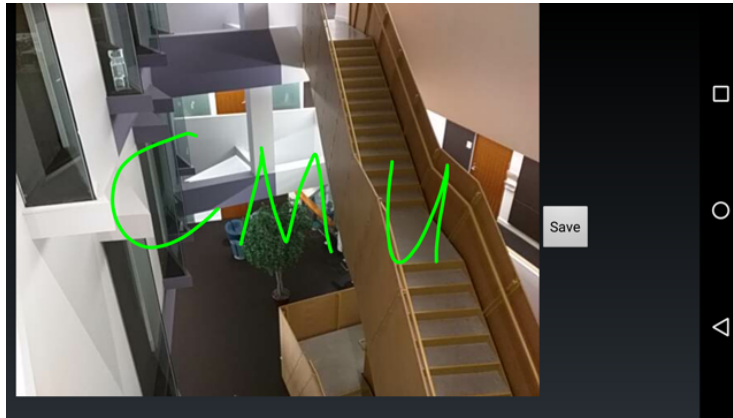


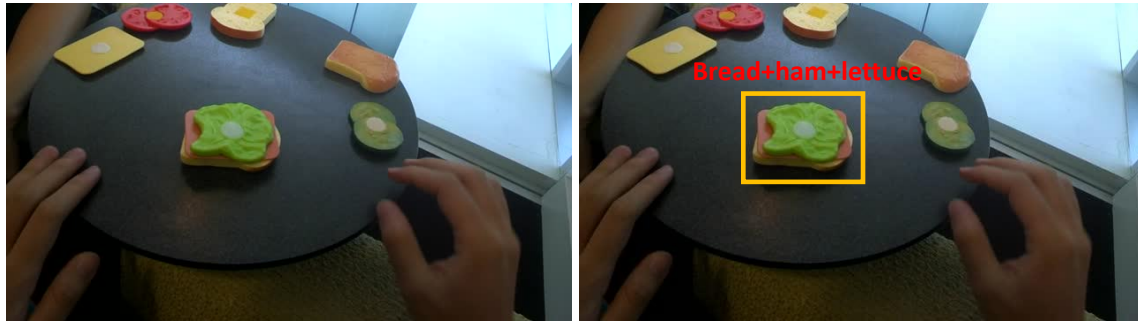
Figure 3.9: Graffiti: Example Screenshot

The second step tries to identify all the pixels associated with the user’s sketches. To do this, I use the difference of Gaussian approach and thresholding to find darker areas in the paper. However, as seen in Figure 3.7(e), this detection can be noisy because of shadows and creases. Therefore, I filter out the noisy parts based on their size, shape, and distance to other components. The result is a clean polygon (Figure 3.7(f)), which is the symbolic representation. As mentioned above, this symbolic representation is fed to the largely unmodified application in place of the pen-based input it was designed for, and the feedback is sent back to be shown on the screen of the mobile device.

3.1.8 Graffiti Assistant

The Graffiti Assistant provides a simple way for people to generate and share augmented information in the form of virtual graffiti on top of real-world objects or scenes. Imagine a user walks in the park and sees a beautiful tree. She can then pull the phone out of her pocket, take a picture of the tree, and then decorate it by drawing with her fingers on the screen. She can also annotate the image with some writings to describe her feelings. After clicking “save”, this augmented picture will be sent to the cloudlet and stored in the database. Other users who walk by will then see the tree decorated in the same way, and receive the text writings, as they point their phone camera towards it. They can then generate their own augmented information in the same way. To prevent users from being overwhelmed by a flood of augmented information, a future version of this application could filter information for each user based on their social contacts, interests, and emotional states. Essentially, this forms a new online social networking application by sharing augmented content based on real-world scenes.

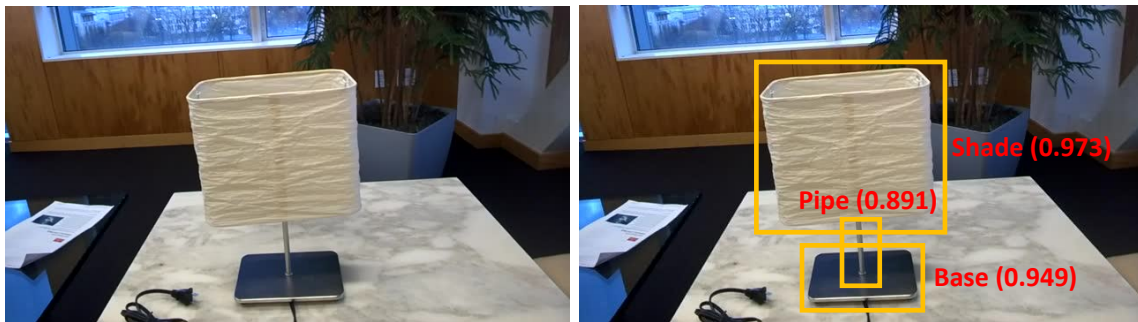
As described above, this application has two modes: the “annotation” mode generates the virtual graffiti content, and the “view” mode displays the graffiti on the camera view in real time. The first mode requires a small extension on the Gabriel client software. With the added “drawing view”, the user can sketch with fingers on top of the captured image (e.g. Figure 3.9). A couple of more buttons help users to switch between different modes.



(a) Input image

(b) Detected object

Figure 3.10: Sandwich: Symbolic Representation



(a) Input image

(b) Detected objects

Figure 3.11: Furniture: Symbolic Representation

The second mode follows the same structure as all other applications. Graffiti uses the SIFT [119] feature vectors to match user captured frame with registered images in the database. Therefore, even if a user is pointing the camera from a slightly different angle or distance, the application can still retrieve the correct augmented information. In addition, Graffiti checks similarity of color histograms to reduce the number of false positives. The extracted feature vectors, as well as the color histogram, are then the symbolic representation of Graffiti. Since the second mode requires no modification of Gabriel software, the measurements in the following sections is using this mode only.

3.1.9 Sandwich Assistant

The Sandwich Assistant helps a cooking novice to make sandwiches following specific recipes. Since real food is perishable, I have developed this application using a children’s food toy that has realistic plastic ingredients that look like bread, lettuce, cheese, tomato, etc. [159]. A YouTube demo of this application can be found at <https://youtu.be/USakPP45WvM>.

As Table 3.1 shows, the symbolic representation is the current sandwich state. For example, “pure bread” is a state, and “bread + ham + lettuce” is another state. A sandwich recipe is described as a list of such states, similar to the task representation in Lego (Section 3.1.6). I train an

object detection model for each of the states in the recipe, and detect these states at runtime using the trained model. The object detection approach I use follows the faster RCNN mechanism introduced in [146]. Figure 3.10 shows an example input frame as well as the object detection result. To speed up the creation of object recognizers for new objects, I use transfer learning [137] to fine-tune a pre-trained object detection model that was originally trained on millions of images from ImageNet [50]. The implementation uses Dlib [105] for bounding box detection, and Caffe [98] for training and applying DNNs. Transfer learning saves me considerable time in labeling data and in training, while also offering high accuracy.

Once the sandwich state is determined, I find its neighboring state in the recipe list and provide guidance to reach that state using visual and verbal guidance. Sometimes a user will make a mistake in making the sandwich (e.g. putting cucumber instead of lettuce). For such cases, I train a model for each possible erroneous sandwich state a user may make and associate appropriate guidance with each state.

3.1.10 Furniture Assistant

When a user buys a piece of furniture from Ikea today, she usually has to assemble it by following the instructions from a paper sheet. Although most of these instructions are very easy to follow, users may still get confused at certain complex steps. What's even more worrying is that sometimes users cannot be sure that they have made all the subtle details correct. The Furniture Assistant I have built offers a simpler and more interactive Ikea assembly experience, using two Ikea pieces as examples, the Magnarp Table Lamp and the Skogsta Stool [94]. Instead of using stationary pictures, the Ikea Assistant delivers short tutorial videos to the mobile device for each assembly step the user needs to complete. More importantly, these videos are delivered just in time for each step by continuously monitoring the user's progress. A video demo on YouTube gives an example of this application: <https://youtu.be/qDPuvBWNIUs>.

The symbolic representation extraction procedure mostly follows that in Sandwich. By using the DNN based faster RCNN approach [146], the application is able to reliably detect the critical pieces in the assembly task, such as the bulb, the base, the shade (from different views), and the buckles. In contrast to Sandwich, there may be multiple different objects being detected from the same frame. For example, in the example image shown in Figure 3.11, the final lamp state is composed of the base, the pipe, and the shade. The relative position of these detected objects matters too (e.g. the pipe must be between the base and the shade). Therefore, the symbolic representation of my Ikea application is the detected objects as well as the relative position among them.

3.2 Quantifying Target Latency Bounds

As stated in Section 2.1.1, all of the above applications have to provide guidance quickly enough so that the users won't get annoyed. While it is relatively easy to measure the system latency of the applications, it is difficult to know if they are fast enough without a reference target. There-

App	Bound Range (tight - loose)	Source
Face	370 ms - 1000 ms	Literature
Pool, Graffiti	95 ms - 105 ms	Literature
Work-out	300 ms - 500 ms	Physical Motion
Ping-pong	150 ms - 225 ms	Physical Motion
Lego, Draw, Sandwich, Furniture	600 ms - 2700 ms	User Study

The numbers are an approximate average across multiple users. They should be treated as guidelines rather than strict limits.

Table 3.2: Latency Bounds of Assistant Applications

fore, in this section, I first derive latency bounds for each of the applications based on their characteristics. While the derived bounds should only be treated as guidelines rather than strict limits, since human cognition and perception exhibit high variability due to individual differences (ability, strategy, etc.), state of the user (e.g. stress or fatigue) [138], and environmental conditions (e.g. lighting) [168], they provide clear guidelines for building cognitive assistance systems that meet user expectations.

I use three general approaches to find the latency bounds of the applications in Table 3.1. I first investigate whether existing published research (e.g., [34, 147]) suggests a reasonable response time bound for our applications. The second approach is to analyze the task from the perspective of first principles of physics. This can work well when characterizing human interactions with physical systems. When these approaches are not applicable, I conduct user studies on my implemented applications. This approach has the advantage of direct relevance, but may be limited in generality [162].

Table 3.2 shows the approach used for each task, and the associated latency bounds. Details about how I derived these bounds are explained in the three subsections that follow. Notice that instead of providing one strict latency bound for each application, I have derived a range of bound parameters. The fast end of this range (the tight bound) represents an ideal target. An application that meets this bound should always be satisfactory to users. At the slow end of the range (the loose bound), the application remains somewhat useful to the user, but this should be avoided if possible. A response that comes later than this may be useless to a user.

3.2.1 Relevant bounds in previous studies

Critical timing for face recognition has been studied extensively in the past. Ramon et al. [144] found that it takes a normal human about 370 ms to recognize a face as familiar, and around 620 ms to realize a face is unfamiliar. This is for a binary classification task (known/unknown). For full recognition of the identity of a face, several studies, including Kampf et al. [102], found normal humans take about 1000 ms. If our Face application aims at providing responses at par with average human performance, it should have a target latency range of 370-1000 ms.

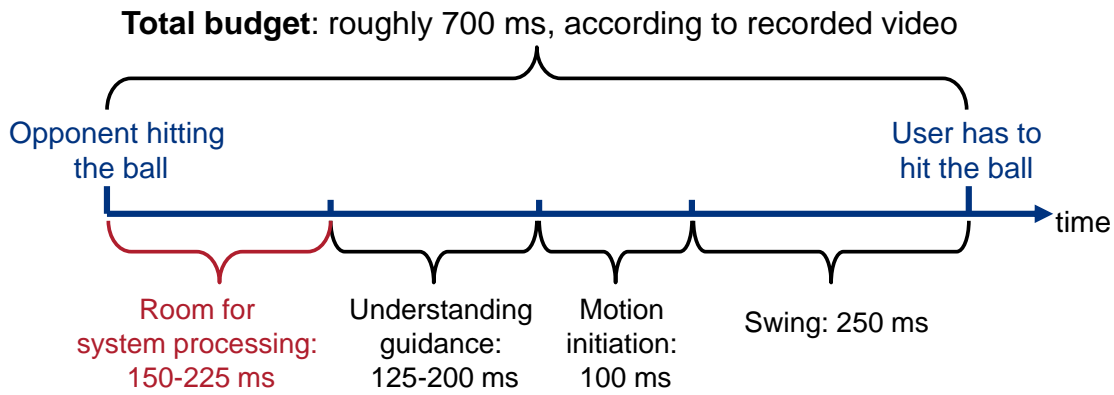


Figure 3.12: Deriving Latency Bound for Ping-pong from Physical Motion

Pool tries to provide continuous feedback to help a user iteratively tune the cue stick position. This requires the feedback to be instantaneous in order to provide the illusion of continuous feedback. This is impossible, of course, but the system should react fast enough that the user perceives this as instantaneous. Miller et al. [125] indicate that users can tolerate up to 100 ms response time and still feel that a system is reacting instantaneously. He also suggested a variation range of about 10% ($\pm 5\%$) to characterize human sensitivity for such short duration of time. Hence, an upper bound of 100 ± 5 ms should be applied to Pool. Similarly, Graffiti aims at providing instantaneous response of virtual graffiti as the user moves the camera. Therefore, the same set of latency bounds could be applied to it.

3.2.2 Deriving bounds from physical motion

Ping-pong has a clear latency bound defined by the moving speed of the Ping-pong ball. Figure 3.12 illustrates how this bound can be derived. From analysis of a videotaped ping-pong game of two novice players, the average time between an opponent hitting the ball and the user having to hit the ball is roughly 700 ms. Within this period, the system must complete processing, deliver guidance to the glass, and guide the user in enough time to react. Ho and Spence [84] show that understanding a “left” or “right” audible guidance as a spatial cue requires 200 ms. Alternatively, a brief tone played to the left or right ear can be understood within 125 ms of on-set [151]. In either case, we need to allow 100 ms for initiation of motion [106]. Our ping-pong video also suggests an average of 250 ms for a novice player to swing, slightly longer than the swing time measured for top-tier professional players [34]. This leaves 150 ms to 225 ms of the inter-hit window for the system processing depending on the audible cue.

In a similar manner, I derive the latency bounds of Work-out as shown in Figure 3.13. I record a video of a user doing sit-ups and analyze the upper bound of latency. The video shows that the user rests for around 300 to 500 ms on average between the point when the completed action can be recognized and when the count information needs to be delivered (starting the next sit-up). Note that the user doesn’t need to wait for or react to the count spoken by the system before starting the next sit-up, so the full 300-500 ms can be spent on system processing. However, with longer delays it is increasingly likely that the user may initiate a sit-up without the benefit

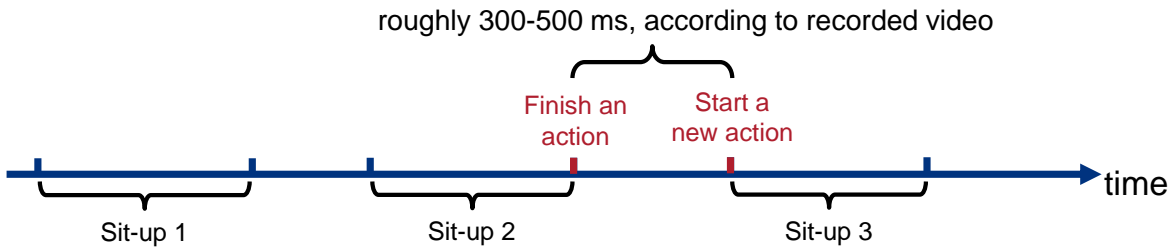
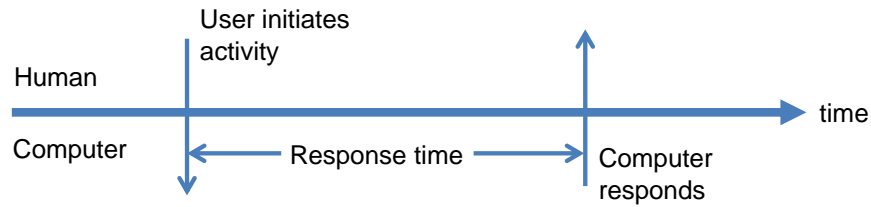
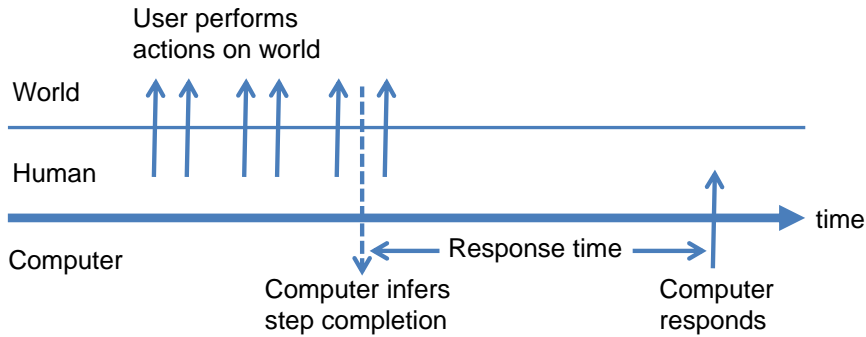


Figure 3.13: Deriving Latency Bound for Workout from Physical Motion



(a) Traditional model [155]



(b) New model in wearable cognitive assistants

Figure 3.14: Simple Model of System Response Time and User Action/Think Time

of feedback.

3.2.3 Bounds for step-by-step instructed tasks

In contrast to the tasks above, there is no well-defined, task-specific latency bound for Lego, Draw, Sandwich, and Furniture where the system provides step-by-step instructions. These tasks are not time-critical, but waiting time between instructions can lead to user dissatisfaction. Previous studies have suggested a two-second limit [125] in human-computer interaction tasks where the waiting cost is not excessive to the user (e.g. clicking a mouse and waiting for a webpage reply). However, such a bound may not be applicable here, due to the difference in the interaction models of traditionally studied systems and our cognitive assistance applications (see Figure 3.14). In the traditional model, a user explicitly initiates an action on the system (e.g. key press), and the system explicitly responds (e.g. display character). For cognitive assistance, the user does not directly trigger actions on the system; rather, she manipulates the surround-

Total number	13
Gender	Male (8), Female (5)
Google Glass proficiency level	Developer (0), Experienced (4), Novice (9)

Table 3.3: Demographic Statistics

ings while performing the current step of the task. The system implicitly infers when the user is done through sensing, and then provides an explicit response (the next instruction). The lack of explicit user initiation can affect perception of delay in a number of ways. For example, the user may feel she is done before the system infers it, so the perceived delay could be longer. On the other hand, without an explicit point of reference, the user may be more tolerant of varying delay.

I conducted a user study to explore bounds for such applications. Specifically, I wanted to answer two questions: 1) How does users' satisfaction vary when the system response time changes? 2) What is the ideal interval between a user finishing a step and receiving the guidance for the next step?

Demographic profile of users

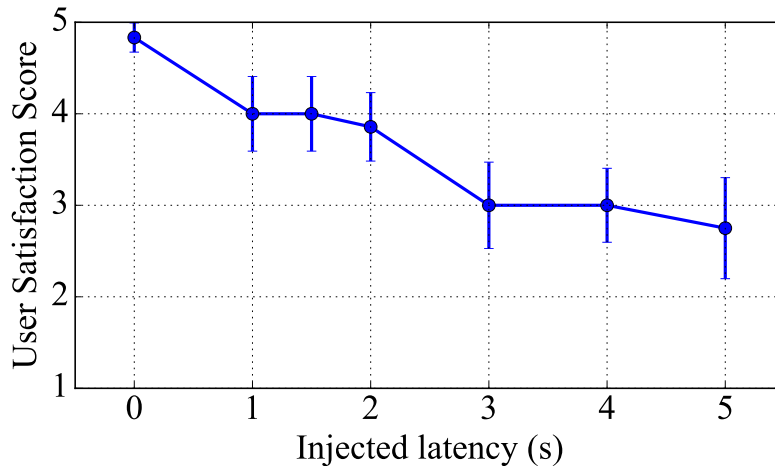
The demographics of the participants are shown in Table 3.3. All of the 13 users were college students (8 male, 5 female). None were experts in Google Glass, but four of them had played with it before. For users not used to the control or display of Glass, I asked them to try basic Glass applications to get familiar with the device before the experiments.

Experiment 1

To address the first question, I asked each user to use Lego Assistant to complete four Lego-assembly tasks, each of which consisted of 7 to 9 steps. An additional latency between zero to five seconds was injected to each task for every participant. This injected latency was randomized, accounting for biasing in results the order of experiments might cause. Each participant was also asked to complete a "warm-up" trial before the four experimental trials to get used to the application.

After each task, the user filled out a questionnaire and had to specifically answer the question: *Do you think the system provides instructions in time?*. The answers were provided on a 1-to-5 scale, with 1 meaning the response is too slow, and 5 meaning the response is just in time.

To control the latency of the experiments, I took a Wizard-of-Oz approach. Instead of relying on any computer vision code to detect the user's state, I streamed the video captured by the user's Glass to a task expert who manually selected the right feedback to send back once the user finished a step. The feedback was delivered to the user after certain additional latency, randomly chosen from 0, 1, 1.5, 2, 3, 4, and 5 seconds for each task. The Wizard-of-Oz approach gave us



User satisfaction: 5 - satisfied; 1 - faster response desired

Figure 3.15: User Tolerance of System Latency

complete control over the response time that users experienced and helped avoid any influence from the application’s imperfect accuracy.

Note that total delay experienced by the users exceeds the delay I injected. The total delay also includes time for network transmissions, rendering output, and reaction times of the task expert. I measured this additional delay to be around 700 ms on average, which I use to adjust our analysis.

Figure 3.15 shows the users’ satisfaction score for the pace of the instructions. When no latency was injected, the score was the highest, 4.8 on average. The score remains stable at around 4 with up to 2 seconds of injected delay. Beyond 2 seconds, the users were less satisfied and the score dropped below 3. These results indicate that application responses within a 2.7 seconds bound (adjusting for the additional delay in the procedure) will provide a satisfying experience to the user.

Experiment 2

To answer the second question, I performed a more open-ended test to determine how soon users prefer to receive feedback. Here, the users were instructed to signal using a special hand gesture when they were done with a step and ready for feedback. A human instructor would then present feedback and the next step both verbally and visually using printed pictures. From recordings of these interactions, I measured the interval between when a user actually completed a step and when she started to signal for the next instruction to be around 700 ms on average. Allowing for motor initiation time of 100 ms [106], this suggests an ideal response time of 600 ms.

Based on these studies, I set a loose bound of 2.7 s and a tight bound of 600 ms for the four step-by-step instructed applications (Lego, Draw, Sandwich, and Furniture Assistant).

3.3 Evaluating Application Performance

In this section, I evaluate the performance of my suite of wearable cognitive assistance applications, in the context of meeting the latency bounds derived in the previous section. Here, latency refers to the end-to-end latency of each application, which includes both networking and processing time. I will use “network RTT” to refer to the portion of latency attributed to network. All of these applications will run on top of the Gabriel platform. While the ideal system setup is to run Gabriel back-end on a cloudlet, I also evaluate the system performance under different configurations. In particular, I compare offloading the application back-ends to different sites, including the public cloud, a local cloudlet over WiFi, and a cloudlet on a cellular LTE network. I also investigate how mobile hardware affects networking and client computation time, and how much different server hardware affects performance. By timestamping critical points of an application, I obtain a detailed time breakdown for it, revealing system bottlenecks and optimization opportunities.

In this section, I answer the following questions:

- How much benefit in end-to-end latency does using a cloudlet offer?
- How does offloading based on cellular/LTE wireless networks compare with those based on WiFi?
- Does the choice of end-user device affect performance?
- How much can extra CPU cores and hardware accelerators in the back-end help?

3.3.1 Experimental Setup

The specifications of the hardware used in my experiments are shown in Table 3.4. Desktop-class machines (the same as used in Section 2.4) running OpenStack [136] are used as cloudlets. For the cloud, I use C3.2xlarge VM instances in Amazon EC2, which are the fastest available in terms of CPU clock speed. This instance type is specifically chosen to match the performance of cloudlets for fair comparison. The Gabriel platform is run in VMs at both the cloud and the cloudlet. In all the experiments, the mobile devices are located on the CMU campus.

For applications that use a GPU (e.g. Sandwich), the GPU-enabled G2.2xlarge instance on EC2 is used. For the cloudlet, a fast GPU is attached to a second machine. Because OpenStack does not provide access to the GPU from a VM, the application code is run on the GPU-enabled machine as a native server. The Gabriel platform continues to run in a VM on OpenStack of the original cloudlet, connecting to the application service over local Ethernet.

I use Nexus 6 phones as stand-ins for high-end wearable hardware in most of the experiments. I also experiment with Google Glass, Microsoft HoloLens, Vuzix M100 Glass, and ODG R7 Glass to demonstrate how client hardware affects latency. They connect to the server over a dedicated WiFi access point using 802.11n with 5GHz support whenever possible. If a device (such as Google Glass) does not support 802.11n WiFi over 5GHz, I use the best WiFi supported by it (e.g. 802.11b/g on 2.4 GHz for Google Glass). All of the devices except HoloLens run a

	CPU/GPU	RAM
Cloudlet	Intel® Core™ i7-3770, 3.4GHz, 4 cores, 8 threads	16GB
Cloudlet (GPU)	Intel® Xeon® E5-1630v3, 3.7GHz, 4 cores, 8 threads NVIDIA Tesla K40 (12GB RAM)	64GB
Cloud	Intel® Xeon® E5-2680v2, 2.8GHz, 8 VCPUs	15GB
Cloud (GPU)	Intel® Xeon® E5-2670, 2.6GHz, 8 VCPUs NVIDIA GRID K520 (4GB RAM)	15GB
Phone	Krait 450, 2.7 GHz, 4 cores	3GB
Google Glass	TI OMAP4430, 1.2 GHz, 2 cores	2GB
Microsoft HoloLens	Intel® Atom™ x5-Z8100 1.04 GHz, 4 cores	2GB
Vuzix M100	TI OMAP4460, 1.2 GHz, 2 cores	1GB
ODG R7	Krait 450, 2.7 GHz, 4 cores	3GB

Table 3.4: Experiment Hardware Specifications

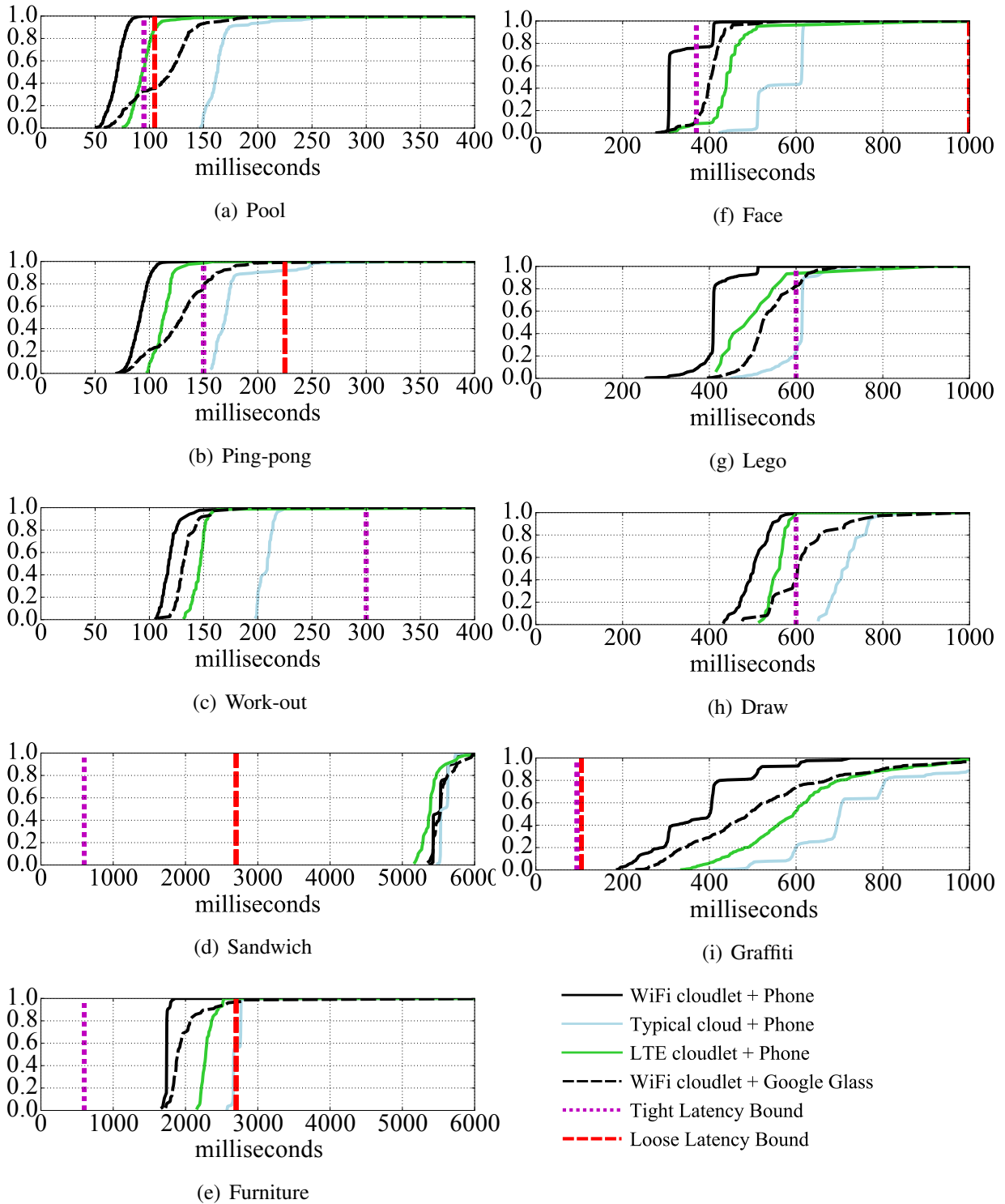
variant of Android, while HoloLens runs Windows 10.

The rest of the setup is identical to that in Section 2.4. The mobile devices are set to capture 640x360 video at 15 frames per second, but send pre-captured frames instead of live frames to have reproducible experiments. Each experiment typically runs for five minutes. For consistent results with some wearable devices, I use ice packs to cool the device to avoid CPU clock rate fluctuation. The Google Glass is also paired with a phone through Bluetooth, as suggested by [15], to get stable WiFi transmission in the experiments.

3.3.2 Response Time with WiFi Cloudlet Offloading

I first evaluate how well my applications perform in the ideal configuration, with the back-end services running on a WiFi-connected cloudlet machine, and a phone as the client device. This serves as the control scenario to which I will compare other system setups that vary particular system parameters.

Figure 3.16 shows the cumulative distribution functions (CDFs) of response times of each of my applications under various system setups. The total end-to-end latency is measured on the wearable device from when the frame is captured to when the corresponding guidance / response is received. This does not include the time to present the guidance (e.g. text-to-speech, image display) to the user. The solid black lines correspond to the setup using the phone and WiFi cloudlets. These curves are almost always the leftmost in each plot, indicating this configuration results in the best latencies among the different system settings.



All LTE experiments except those for Graffiti and Furniture are set up with help from Vodafone Research, as explained in Section 3.3.5. The setup was no longer supported when these two applications were developed. Instead, the LTE experiments for them were conducted in Living Edge Lab [95].

Figure 3.16: CDF of Application Response Time

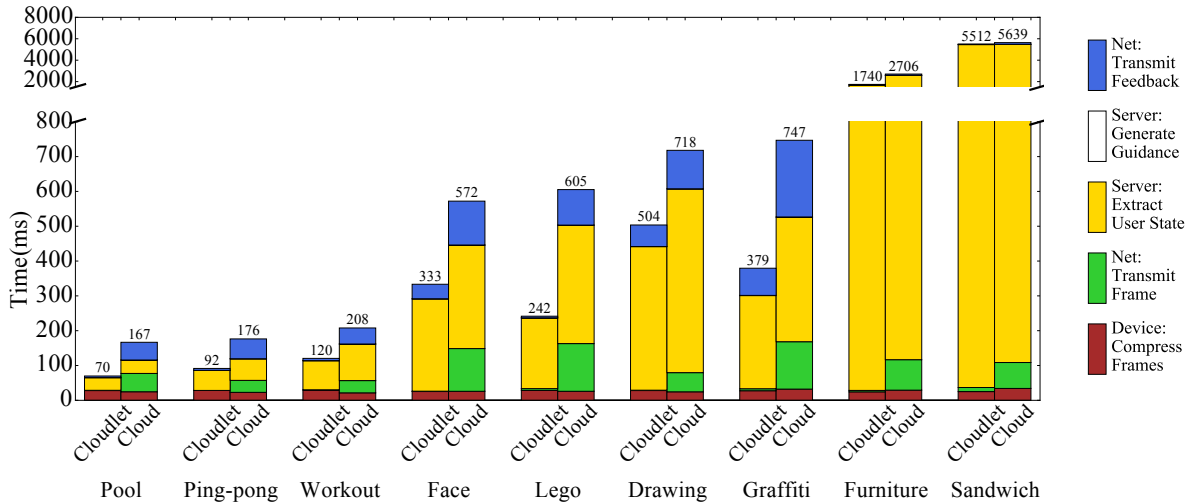


Figure 3.17: Breakdown of Application Response Time

The shape of the CDF lines also demonstrate variation in application response times. Some of this is a result of network jitter, but most is due to processing time variation for different video frames. Between the different applications, the latencies can vary significantly, e.g., Sandwich is almost 100x slower than Pool. In general, except for Sandwich and Furniture, all of the applications achieve latencies on the order of a few hundreds of milliseconds.

Next, I compare the response times of all of the applications with the latency bounds derived in Section 3.2. The dashed purple and red vertical lines indicate the tight and loose bounds. As the figure shows, seven of the nine applications successfully meet at least the loose latency bounds. Five of them can also meet the tight bounds. These include applications like Pool and Ping-pong which have very stringent latency bounds of less than 200 ms. Face Assistant easily meets the loose latency bound, and comes close to the tighter bound of 370 ms. Not surprisingly, the applications that fail to meet their bounds are Sandwich, and Graffiti, which involve very heavy computation such as DNNs.

3.3.3 Time breakdown of applications

In addition to the total end-to-end latency analysis, I also log timestamps at major points in the processing pipeline to further break down where the time is spent for each application. In particular, I measure time taken for five main functions: (1) Compressing a captured frame (MJPEG), (2) Transmitting the image to the server, (3) Extracting the symbolic representation on the server, (4) Generating guidance based on the symbolic representation, and (5) Transmitting guidance back to the Glass. To calculate the time spent on network transmission, time synchronization between the server and the client is needed. Therefore, the mobile device exchanges timestamps with Gabriel server to synchronize time at the beginning of each experiment.

The left bar of each application in Figure 3.17 shows the time breakdown when offloading to the cloudlet. Several interesting insights can be gained from this figure.

- First, for most of the applications, relatively little time is spent on network transmission. This is the advantage that cloudlets promise – cloud-like resources with low latency and high bandwidth, resulting in low network overheads. As a result, a majority of time is spent on server computation, reflecting the complexity of algorithms being used. The server-side computation is thus a clear bottleneck, and should be the focus for further optimization.
- Second, except for Draw, guidance generation takes negligible time. Although this non-trivial step needs advanced task-specific knowledge, once the symbolic representation is available, guidance generation is computationally cheap. For example, for Pool, translating the cue, ball, and pocket positions to aiming guidance needs pool-specific knowledge of how to aim, but computation-wise it is simply angle calculation based on three points and a small table lookup.
- Third, some of the networking time for transmitting feedback is larger than expected. For example, in Face, transmitting the few bytes of guidance to the client device (the blue component) even takes longer time than streaming the video frame to the server (the green component). This is because the mobile devices usually employ aggressive power management of the radio link. Thus, the time for network receiving include overheads of transitioning out of low-power states, which can sometimes be hundreds of milliseconds.
- Finally, there is a big variation in the total response time (number on top of each bar) across different applications. This shows the diversity of algorithms being used for different applications. Among them, Sandwich and Furniture remain the big outliers, with far higher response times than the other applications. Their time breakdown are dominated by the symbolic representation extraction step, which averages over five seconds, due to the use of a computationally expensive deep neural network.

3.3.4 Benefits of Cloudlets

Is offloading to a cloudlet necessary to achieve such low response time, or does cloud offloading suffice? I repeat my experiments, using an Amazon AWS server to host the application back-ends. I use an instance in AWS-West to represent a typical public cloud server, as the round trip time (RTT) from my test site (CMU campus) to AWS-West corresponds well to reported global means of 74 ms [112]. Note that I did not choose EC2-east because the campus lab has unusually good connectivity to EC2-East (8 ms latency, 200 Mbps BW), which should not be considered typical.

The solid blue lines in Figure 3.16 corresponds to response time when offloading to EC2-West. In general, offloading to cloudlets is clearly a win for my benchmark suite. Looking more closely, cloud offload almost always incurs an additional 100 to 200 ms latency compared to using a cloudlet. For Pool, the 90th percentile latency of cloud offloading is more than 2x that of cloudlet offloading. This additional latency significantly reduces the chances of meeting the latency bounds derived earlier. When offloading to the cloud, Pool can no longer meet even its loose latency bound, and Ping-pong and Furniture are on the edge of missing their loose bounds. Face, Lego, and Draw meet their loose latency bounds with the cloud, but are not even close to meeting their tight latency bounds. Work-out remains to be the only application that still meets

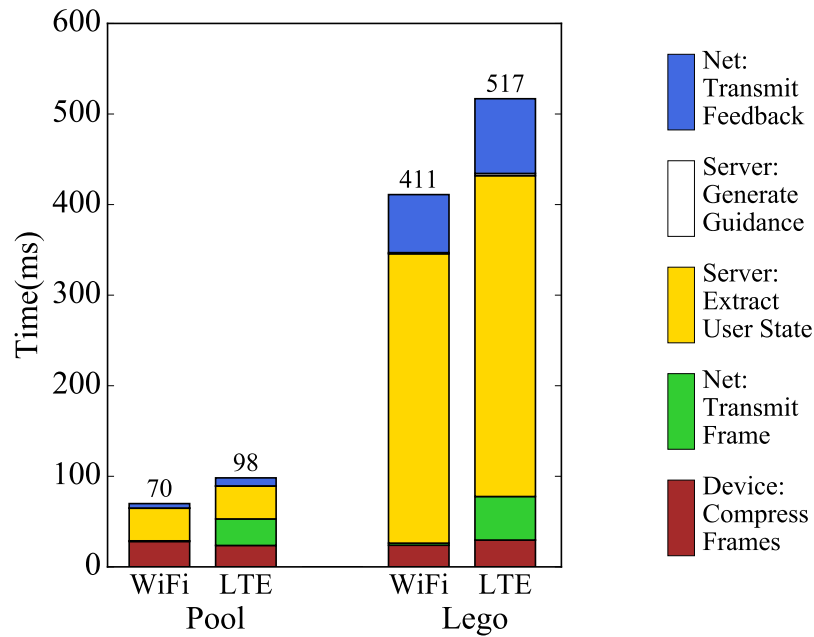


Figure 3.18: Latency Breakdown - WiFi vs. LTE Cloudlets

its tight latency bound. So, in order to meet the tight latency bounds for many applications, or even meet the loose bounds for Pool, Ping-pong, and Furniture, it is necessary to use cloudlets.

How much of the performance difference is due to the network versus other differences between the cloud and cloudlet? The right bar of each application in Figure 3.17 reveals the time breakdown when offloading to the cloud. As we see, the performance difference between cloudlet and cloud, in most cases, comes from network time. The time spent on network transmission of both frames and guidance increases dramatically as the offloading site gets farther. In the cloud case, up to 50% more time is spent on network transmissions.

The performance gain in using a cloudlet sometimes also comes from the better hardware we have on the cloudlet. This can be seen from the breakdown of Draw and Furniture in Figure 3.17 where the computational time on a cloudlet is smaller than that in a cloud. This is because the CPU on the cloudlet has a higher clock-rate which will benefit the complex processing. The fact that a user has better hardware on a cloudlet than in the cloud may not be a rare case, especially for home-based cloudlets. For cloud offloading, even if a user is willing to spend extra money, she won't get new hardware until the cloud providers decide to do so. On the other hand, a cloudlet that is under the user's control (e.g. in the user's home) can be upgraded overnight.

3.3.5 4G LTE vs. WiFi for First Hop

The initial research efforts on cloudlet offloading focused on cloudlets connected to local WiFi networks. This connectivity provides excellent bandwidth and low latency to mobile devices on the local WiFi network. However, this is not the only place to deploy cloudlets. In particu-

lar, as part of a larger push toward network function virtualization (NFV) and software defined networking (SDN), the telecommunications industry is pursuing plans to install general-purpose compute infrastructure within cellular networks [35], enabling LTE-connected cloudlets. How much does LTE as the first wireless hop affect application latency?

To answer this question, I need access to LTE-connected cloudlets. As LTE-based cloudlets are not yet commercially deployed, our research group has built a prototype system in the lab. With assistance from Vodafone Research, we have set up a small, low-power (10 mW) 4G LTE network in our lab, under an FCC experimental license. A cloudlet server connects to this network's eNodeB (base station) through a Nokia RACS gateway, that is programmed to allow traffic intended for the cloudlet to be pulled out through a local Ethernet port without traversing the cellular packet core. This local break-out capability is expected to be a core feature of future deployments of cellular cloudlet infrastructure.

I measure the response times of our suite of applications using the cloudlet connected to the eNodeB to host the application back-ends, and using the phone connected to the lab LTE as the client device. As far as I am aware, this thesis is the first to evaluate the performance of LTE-connected cloudlets with complete mobile applications.

The solid green lines in Figure 3.16 show the CDFs of system response times using the LTE-connected cloudlet configuration. These lines are consistently to the right of the lines corresponding to WiFi-connected cloudlets, with about 30 milliseconds difference. Excluding Aperture and Furniture, which are using a different LTE setup as explained in Figure 3.16, the relative difference of the 90th percentile latency is up to 33.8% (Pool). These differences are consistent with expectations, as 4G LTE has longer network latency (as measured by ping), and lower bandwidth compared to WiFi. As shown in Figure 3.18, I also plot the time breakdown comparing WiFi and LTE scenarios for two applications: Pool, which is the most interactive one, and Lego, which is one of the slower ones. The figure confirms that most of time difference is due to increase in network transmission times in LTE.

Overall, although LTE cloudlets are at a disadvantage when compared to WiFi cloudlets, they do provide reasonable application performance. In particular, Figure 3.16 shows that the latencies are better with LTE cloudlets than with cloud-based execution for all of the applications studied. In comparing with the target latencies, the applications have similar chances of meeting them as when WiFi is used. In particular, only Pool and Face can no longer meet their tight bounds, but are still close. Thus, I believe that LTE cloudlets are a viable computation offloading strategy and can provide significant benefits over the default of cloud offloading. Furthermore, if 5G lives up to expectations of greatly reduced radio network latencies, and the promises of greater bandwidth, the performance with cloudlets in the cellular network will only improve.

3.3.6 Effect of Better Client Hardware

Since most processing is offloaded to the Gabriel back-end, one expects the choice of mobile device to have little impact on end-to-end latency. To confirm this intuition, I repeat the latency measurements of my applications running on a WiFi cloudlet, but replace the phone with Google Glass as the client device. The results are shown by the dashed black lines in Figure 3.16.

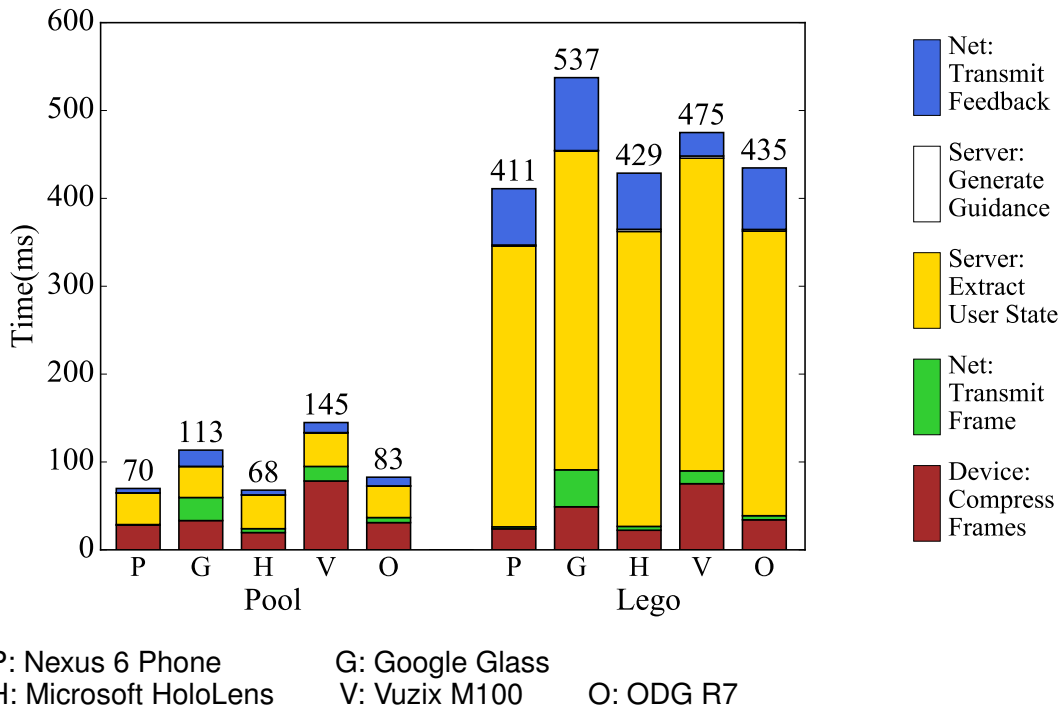


Figure 3.19: Latency Breakdown - Client Hardware

Surprisingly, using Glass has a profound effect on the latencies, increasing them across the board for all applications. For Pool, the 90th percentile latency increases by 80% compared to the results with the Nexus 6 phone. Furthermore, there is greater variability in the response times, as indicated by the shallower slope of the CDFs. As a result, only one out of all the applications can meet its tight latency bounds.

To further investigate this surprising result, I capture the time breakdowns for the applications with five different client devices: the Nexus 6 phone, Google Glass, Microsoft HoloLens, Vuzix M100 Glass, and ODG R7 Glass. The measured time breakdowns are shown in Figure 3.19. I show results for the same two applications as in Section 3.3.5. For Pool, I see that the average response time (total bar height) varies significantly across client devices. A part of this difference is due to differences in processing (compression) speed on the client device. In particular, the Vuzix device takes significantly longer time than the others on the compression step. The bulk of the observed differences across mobile devices can be attributed to different network transfer speeds. For example, Google Glass only has 2.4 GHz 802.11b/g technology, but the Nexus 6 phone uses 5 GHz 802.11n to fully leverage the WiFi bandwidth to minimize transfer delay. For Pool, where the network transfer delay is significant, the total response time is reduced by nearly 40% by switching from Google Glass to the Nexus 6 phone. The Vuzix device has networking that falls between Glass and the phone, while the HoloLens and the ODG device appear to be more capable wearable devices, generally matching the phone in all attributes.

Similar differences in compression and network transmission occur for Lego. However, since the server computation takes much longer time here and is not affected by the choice of client

App	1 core	2 core	4 core	8 core
Lego	415.0 (65.4)	412.5 (53.3)	420.3 (62.4)	411.0 (38.9)
Sandwich	12579.5 (60.8)	7237.8 (49.8)	6657.6 (1207.2)	5312.3 (107.9)

The numbers in parenthesis are standard deviations of sample points in 5-min runs.

Table 3.5: Mean Latency (ms), Varying Number of Cores

hardware, the change in the overall system response time is relatively smaller than for Pool.

3.3.7 Effect of Cloudlet Hardware: Varying Number of Cores

So far, I have investigated how system configuration can significantly influence the response times of the applications, primarily due to differences in network transmission and processing times on client hardware. The one aspect that remains stubbornly unchanged, and the bottleneck for most applications, is the time required for processing at the server. In the following sections, I look at systematic approaches to reduce the latency due to server processing, by looking at simply applying more CPU cores, and then exploring if hardware accelerators can help. In Chapter 4, I will propose a novel technique of applying multiple blackbox algorithms in parallel to obtain results faster with little impact on accuracy.

One advantage to an elastic, cloud-like infrastructure is that provisioning additional resources to a problem is a relatively simple undertaking. In particular, it is easy to allocate more cores to running the application back-ends. Although it is generally easy to use such additional resources to improve the throughput of a system (e.g., one can simply run multiple instances of a service and handle multiple front end clients simultaneously), using additional resources to improve latency is usually a much harder task. This is because the applications may not be structured to benefit from additional cores, due to inefficient implementation or due to fundamental constraints of the algorithms being used. It may be possible to rewrite an existing application to better exploit internal parallelism using multiple cores, but this may require fundamentally different algorithms to allow greater parallelism.

Can extra cores help improve the latencies achieved by my suite of applications? Here I evaluate two of the applications with large server processing components to see how their performance varies with the number of virtual CPUs provided. The results are summarized in Table 3.5. The Lego implementation is largely single threaded, but performs some operations that are internally parallelized in the OpenCV library. Therefore, as expected, the system response time remains almost unchanged as the number of cores is varied. The Sandwich application, on the other hand, is based on a neural network implementation, which can be executed by several well-parallelized linear algebra libraries. Therefore, it is able to run faster as the number of cores increases. However, even here, we see diminishing returns – most of the gains in latency are obtained when the number of cores is increased to two from one. This is likely due to the fact that on these specific computations, the implementation of the linear algebra libraries used were not able to efficiently leverage the full parallelism provided by the system. Therefore, further increasing the core count only modestly affects latency. Even with eight cores, latency is unacceptable at about

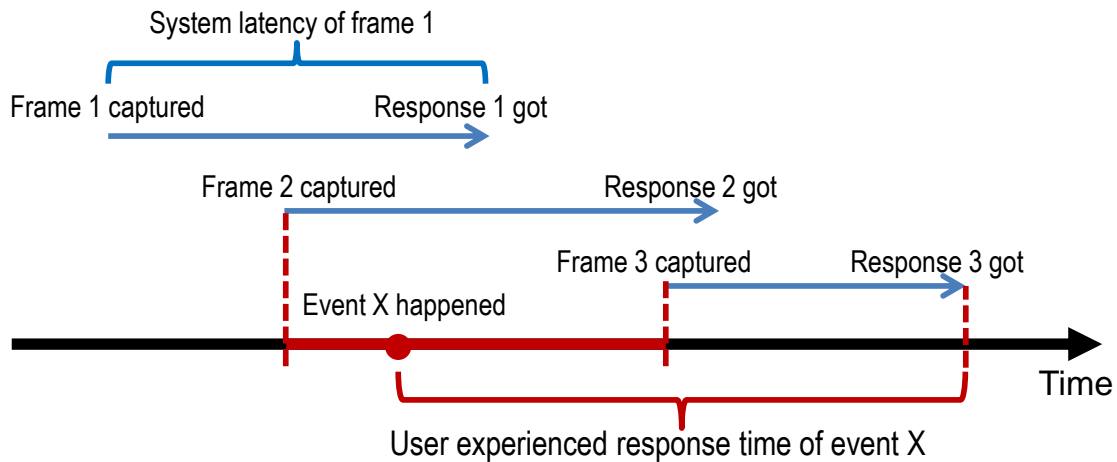


Figure 3.20: Difference Between System Latency and User Experienced Response Time (UERT)

five seconds.

3.3.8 End-to-end Latency vs. User Experienced Response Time

As mentioned in the previous section, another way to leverage multiple processing units on the server (e.g. CPUs) is to feed different data units to different processors by recognizing that many image processing tasks naturally exhibit coarse grain parallelism. For example, one can feed different frames or video chunks to different cores for concurrent processing. Unfortunately, this alternative of leveraging multiple cores/machines to exploit coarse-grain data parallelism typically only improves throughput, not end-to-end latency.

However, I show here that a simple scale-out can help to improve system response time of wearable cognitive assistance applications from a user's perspective. I start by looking closely at the definition of system response time. The system response time (or end-to-end latency) in my discussion so far is the latency between the point when a frame is captured and the point when the response associated with that frame returns. In reality, however, user experienced response time can be longer. This is because the system can only process a subset of frames generated by the camera at intervals determined by the available processing resource and speed of computation. When a user initiates an activity or has finished a step, the system may be busy and not capture the relevant frame immediately. For example, Figure 3.20 shows the capture and guidance return time of three frames. The length of the three blue arrows indicates the system latency. However, if any event X happens after frame 2 is captured but before frame 3 is captured (marked as red in the figure's timeline), the user has to wait until the point when the response of frame 3 returns to get helpful guidance. The response time a user has to wait in this case is longer than the system latency of any frame.

On average, the system waits for half of the interval time before a frame is picked up. The user experienced response time then includes the system latency of one frame, and the average

App	Face	Pool	Workout	Ping-pong	Lego	Draw	Sandwich
Latency (ms)	496	113	134	129	517	619	264
UERT (ms)	734	175	203	190	606	677	406

Figure 3.21: Comparison of System Latency and UERT

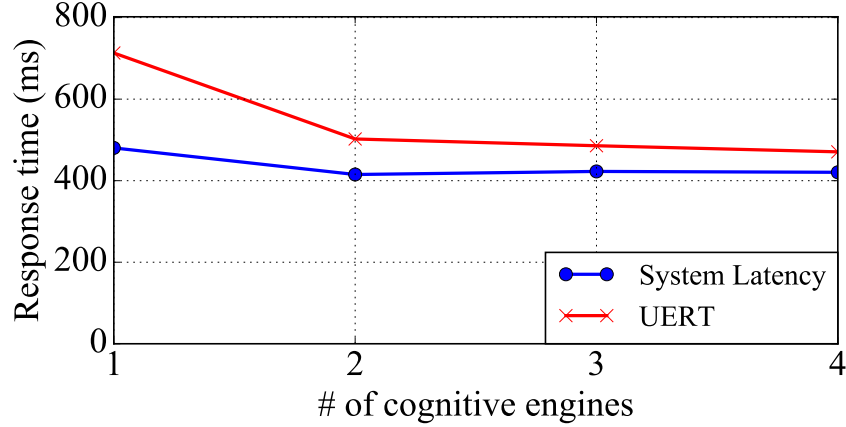


Figure 3.22: Improvement of UERT with Different Number of Cognitive Engine Instances (Face)

waiting time. Therefore, I define the User Experienced Response Time (UERT) as

$$UERT = \frac{1}{n} \sum_{i=1}^n (t_{end}^i - t_{start}^i) + \frac{1}{2} (t_{start}^i - t_{start}^{i-1}) \quad (3.1)$$

where t_{start}^i is the time frame i is captured, and t_{end}^i is when its response is got. Therefore, $t_{end}^i - t_{start}^i$ denotes the time to process frame i , and $\frac{1}{2}(t_{start}^i - t_{start}^{i-1})$ represents the average waiting time before processing the frame.

I measure UERT of my suite of applications according to equation 3.1. Figure 3.21 compares the UERT with average latency for each frame. Clearly, UERT is always larger than system latency, and for most cases it is around 50 percent larger than average latency.

Next, I show how a simple scale-out of existing applications could improve UERT. This approach works because a higher throughput helps to reduce the interval the system picks up frames. I use Face as an example to demonstrate this approach. To increase system throughput, I attached multiple Face modules as cognitive VMs into Gabriel’s publish-subscribe backbone. The Control VM then distributes the input frames to different Face VMs whenever they are idle. As we can see in Figure 3.22, when the number of cognitive engines attached increases, system latency does not change much. However, UERT becomes smaller and closer to system latency, since the system’s processing interval becomes smaller.

In conclusion, a simple scale-out of existing applications does help to reduce UERT. A recent work [20] has offered similar insight in how increasing throughput can benefit application latency. In particular, when sufficient compute engines are used, UERT can be as small as the system latency, but not smaller. Therefore, the rest of the dissertation, especially Chapter 4, will

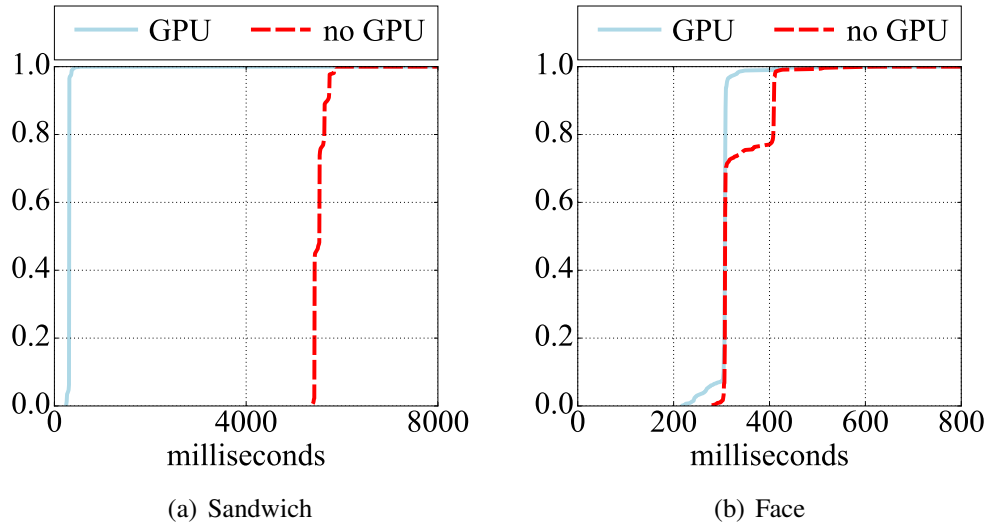


Figure 3.23: CDF of Latency with and without Server-side Hardware Acceleration (GPU)

focus on improving system latency, which will automatically improve UERT as well.

3.3.9 Effect of Cloudlet Hardware: Accelerators

To study the benefit of using specialized hardware, I experiment with a modern GPU as a proxy for hardware accelerators that may be available in cloud(let) computing environments of the future. I study the effect of enabling and disabling access to the GPU using Sandwich and Face, both of which use GPU-optimized neural network libraries. As Figure 3.23(a) shows, Sandwich benefits significantly. This is because it uses a very deep neural network, and almost all of the processing is in the accelerated code paths that leverage the GPU. On the other hand, Face gets very little benefit (Figure 3.23(b)), as the accelerated computations form a much smaller share of the overall computation.

As these results show, there is a large potential benefit to exposing server-side hardware accelerators to applications. However, the benefits are application-dependent, so these need to be weighed against the very real costs of this approach. Beyond the monetary costs of additional hardware (e.g., high-end GPU, or FPGA), management of a cloudlet becomes more complicated, as sharing such resources in a virtualized multi-tenant system is still a largely unsolved problem. Furthermore, applications may be tied to very particular hardware accelerators, and may or may not work well with other hardware. These compatibility issues threaten to fragment the space and limit the vision of ubiquitously available infrastructure to run any application. Finally, the relative benefits of hardware acceleration is constantly evolving as new algorithms are developed. For example, a recent work [145] uses approximation techniques to process deep neural networks at interactive speeds without hardware acceleration and very little loss in accuracy.

	Pool	Graffiti	Workout	Ping-pong	Face	Lego	Draw	Furniture	Sandwich
Bound Range w/ Phone	95-105		300-500	150-230	370-1000	600-2700			
WiFi Cloudlet w/ Phone	80	516	131	102	410	455	546	1755	5708 w/ GPU: 308
WiFi Cloud w/ Phone	173	1054	216	192	615	619	767	2766	5640
LTE Cloudlet w/ Phone	107	830 ¹	154	123	498	578	583	2517 ¹	5677
WiFi Cloudlet w/ Glass	144	799	146	164	435	618	725	2588	5754

Table 3.6: 90th Percentile System Latency vs. Latency Bounds (in milliseconds)

3.3.10 Summary

Table 3.6 summarizes the latency bound ranges derived for each application, as well as the 90th percentile latencies achieved in a variety of configurations. These latency numbers can be derived from Figure 3.16 by intercepting the CDF lines with the 90% horizontal line. I color-code these numbers using green, orange, and red when both, only loose, or neither latency bounds are met.

Clearly, using a WiFi-connected cloudlet has the advantage in providing fast system responses and the highest chance to meet the latency bounds. However, these widely distributed WiFi cloudlets are expensive to deploy and manage. The second best choice is to use a 4G/LTE-connected cloudlet. These cloudlets can be easier to build and maintain, and still offer a large chance to meet the latency bounds. If no cloudlets are available, reaching the cloud may still help, but are only useful for less latency sensitive applications.

Hardware improvement of the wearable device also significantly benefits user experience. As shown in the table, replacing the phone with Google Glass as the client device decreases user experience to a degree similar to substituting the cloudlet with the cloud. However, when a modern mobile device such as a phone or HoloLens is used, the time spent on networking and client processing is very small, leaving most part of the latency attributed to server processing. Therefore, further improvement on client hardware will not significantly improve end-to-end latency. However, efforts to control thermal dissipation on a mobile device is still greatly needed to achieve stable performance and user comfort.

Finally, Table 3.6 shows that using a GPU for Sandwich could reduce its end-to-end latency below the latency bounds. I have also shown in earlier sections that simply adding more CPU cores to the system could benefit many applications. Therefore, a multi-tenant cloudlet that supports multiple applications and users simultaneously would need to be computationally powerful. In addition, accelerators for computer vision, video processing, or vector computation can provide significant boost to cognitive assistance applications, so they should be considered in future cloudlet deployment.

¹The LTE setup for these two applications are different from other ones, as explained in Figure 3.16.

Chapter 4

Speeding up Server Computation: A Black-box Multi-algorithm Approach

In the previous chapter, I have shown that server computation time is the bottleneck for end-to-end latency in most wearable cognitive assistance applications. Simply adding cores or a hardware accelerator could help to improve latency for some of them, but may not work for the others. In general, simply throwing additional resources at the problem has limited benefits, which depend on the exact algorithm and implementation used in the application. Even with all these resources, the applications may still not be fast enough to provide satisfactory user experience.

So what options are left for improving application latency? If one could devise a new algorithm that can solve the perception problem more efficiently, or re-implement the original algorithm in a way that takes better advantage of the hardware available, then this could reduce latency. However, this usually requires an algorithmic breakthrough, or fundamental rewriting of the existing code. Hoping for such progress is not a reasonable strategy. On the other hand, if one could live with a loss of accuracy, there are often relatively cheap alternative algorithms that are fast, but less accurate. Can we use the cheap algorithm to improve the performance of the system, but without suffering accuracy loss?

In this chapter, I propose a novel meta-algorithm that combines multiple algorithms that address the same problem but with different speed and accuracy characteristics. By leveraging temporal locality of accuracy in a video stream, the multi-algorithm approach dynamically switches among different algorithms, leading to significantly higher speed on average, with little sacrifice in accuracy. One big advantage of this approach is its black-box feature: no modifications to the original algorithms are needed. Therefore, it is straightforward to apply this approach to different applications or computer vision tasks even without access to their source code.

The rest of the chapter is organized as follows. Section 4.1 describes the intuition behind this approach. Section 4.2 then explains how this algorithm works, and how it can be implemented on the Gabriel platform. Section 4.3 experiments with this approach using Face in an offline setting to explore the effect of different algorithmic parameters. Evaluations on the Gabriel platform are then shown in Section 4.4 using three of my benchmark suite applications. In

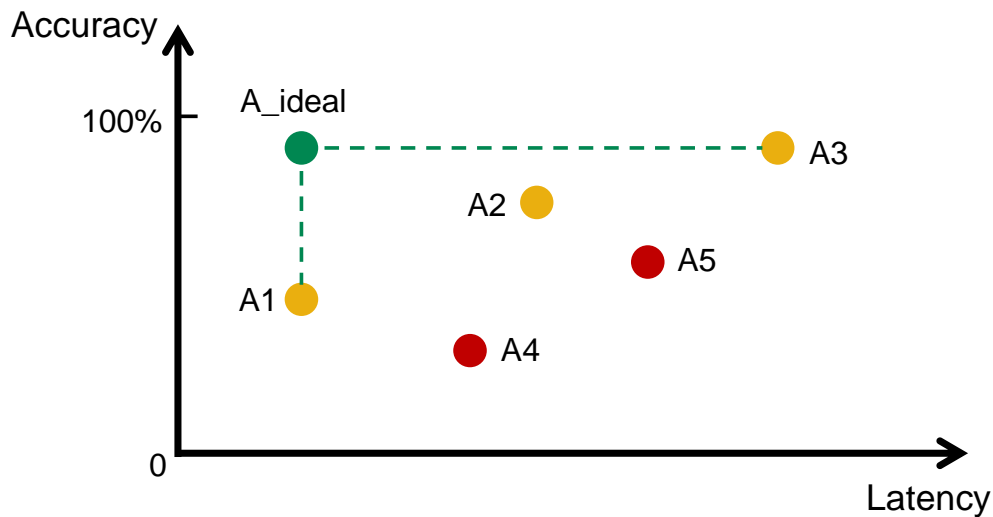


Figure 4.1: Tradeoff Between Speed and Accuracy in Computer Vision Algorithms

addition, Section 4.5 introduces how this approach can be used outside the Gabriel context, and demonstrates its value using a new application. Finally, Section 4.6 presents more discussions on the understanding and limitation of this approach.

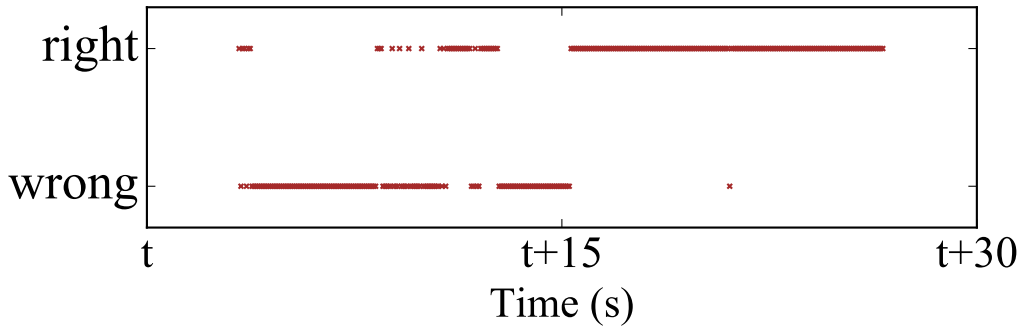
4.1 Background: Two Key Insights

4.1.1 Speed-accuracy Tradeoff in Computer Vision Algorithms

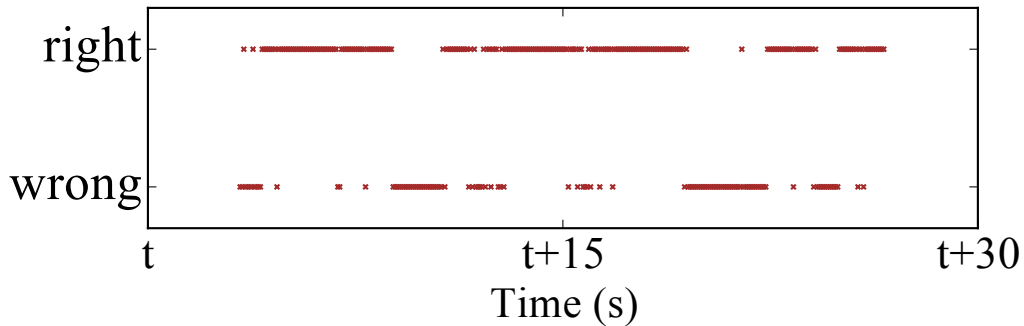
In the computer vision community, accuracy is the chief figure of merit. “Better” algorithms usually are equated with more accuracy, regardless of the computational requirements. Thus, over the years, newer, often more complex and computationally intensive techniques have been devised to solve particular problems at ever-increasing levels of accuracy as measured through standardized data sets. The state-of-the-art at any given time is often quite slow due to computational complexity.

As a result, there are often many different algorithms to solve any common computer vision task. The state-of-the-art method from a few years ago is typically less accurate than today’s best, but is also often less complex and faster to execute on today’s hardware. For example, in an object classification task, a HOG+SVM approach is one of the simplest and fastest algorithms, but only gives modest accuracy. In the meantime, the most recent deep neural network (DNN) based approaches can provide much higher accuracy, but are also much more computationally expensive. Thus, there exists a natural tradeoff between accuracy and speed that one should consider when selecting an algorithm to use.

Figure 4.1 illustrates this tradeoff by placing each algorithm as a point in the latency-accuracy tradeoff space. The three yellow points (A_1 , A_2 , A_3) represent three decent algorithms that may exist today, each achieving a different tradeoff between accuracy and speed. Among them, there



(a) Object detection using Faster RCNN [146])



(b) Face recognition using Fisher algorithm [29])

Figure 4.2: Temporal Locality of Algorithm Correctness

is no “best” algorithm, as no one is both more accurate and faster than anyone else. In general, if an algorithm is positioned in Figure 4.1 such that no other algorithms fall on the top left part of it, this algorithm is better than anyone else in either speed or accuracy. Even for the algorithms such as *A4* and *A5* that have lower speed and accuracy than others, there might still be specific categories that they are good at (e.g. they might be the best algorithm to detect “cup”, though really poor at detecting “egg”). Therefore, all these algorithms could be candidates for a particular application. Depending on the use case and a user’s tolerance to delay and error, one of them may be the most suitable.

Of course, none of these algorithms are perfect, as each of them sacrifices either speed or accuracy. Thus, the goal of this chapter is to develop methods to achieve the green point in Figure 4.1, which is as fast as the fastest available algorithm, and as accurate as the most accurate one. In later sections, I’ll demonstrate that what the multi-algorithm approach can achieve is quite close to this point.

4.1.2 Temporal Locality of Algorithm Correctness

Another key insight behind this approach is that the mistakes an algorithm makes are not simply random events. Rather, they are correlated to the environment or the scene to which the algo-

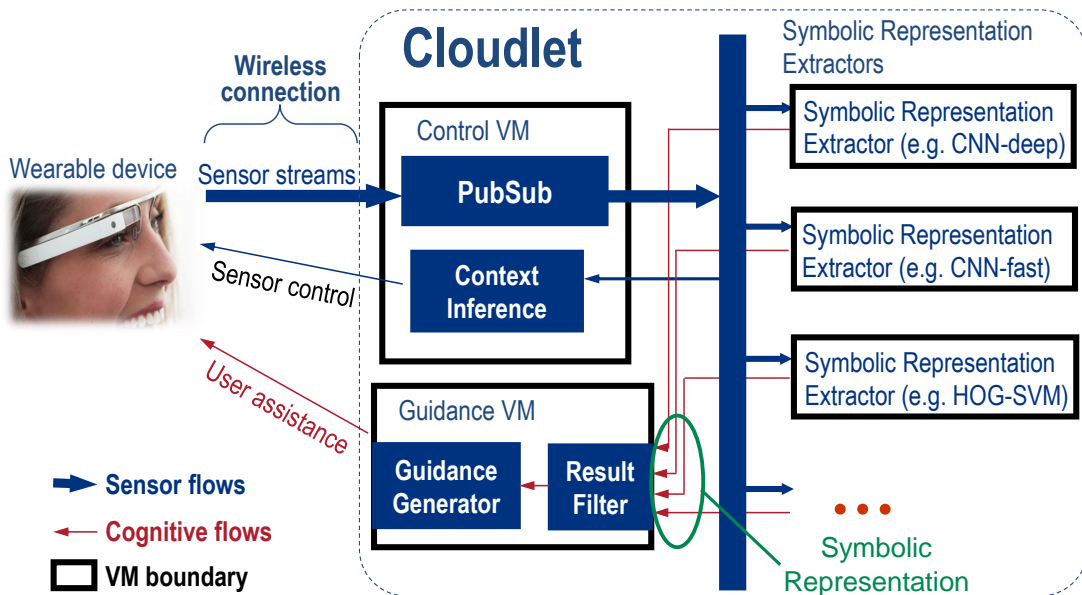


Figure 4.3: Adapted Gabriel Architecture for Multi-algorithm Approach to Reduce Latency

rithm is applied. For example, the lighting conditions, image exposure level, background clutter, and camera angle can all affect the accuracy of an algorithm. More accurate algorithms are typically more robust to changes and work well over a larger range of conditions than less accurate ones. However, because accuracy is tied to the physical scene characteristics, computer vision algorithms exhibit *temporal locality* of accuracy over sequences of video frames, as the scene conditions rarely change instantaneously. Thus, when conditions are such that the algorithm is generating accurate results, one can expect it to continue generating good results in the near term. Similarly, when an algorithm starts to generate incorrect results, it is likely it will not generate accurate results for the following frames.

Figure 4.2 demonstrates this temporal locality in two example algorithms. Figure 4.2(a) is an DNN-based algorithm for object detection, and Figure 4.2(b) is a face recognition algorithm using Fisherface features [29]. Over a sequence of frames, I plot when the algorithm generates results matching known ground truth. Though sometimes wrong, the algorithms tend to produce long runs of outputs that are all right or all wrong. This proves that temporal locality is a real phenomenon that can be exploited in a computer vision system.

4.2 A Black-box Multi-algorithm Approach

Based on the key insights mentioned above, I propose a novel approach for combining different algorithms to result in both high accuracy and low average latency by utilizing additional hardware. The essential idea is to run different algorithms concurrently, using the slower (but more accurate) ones to dynamically evaluate the real-time accuracy of the faster (but less accurate) ones. If a fast algorithm has been accurate for a period of time, the next result it generates will be trusted and used. If not, the system will wait for the more accurate algorithm to finish.

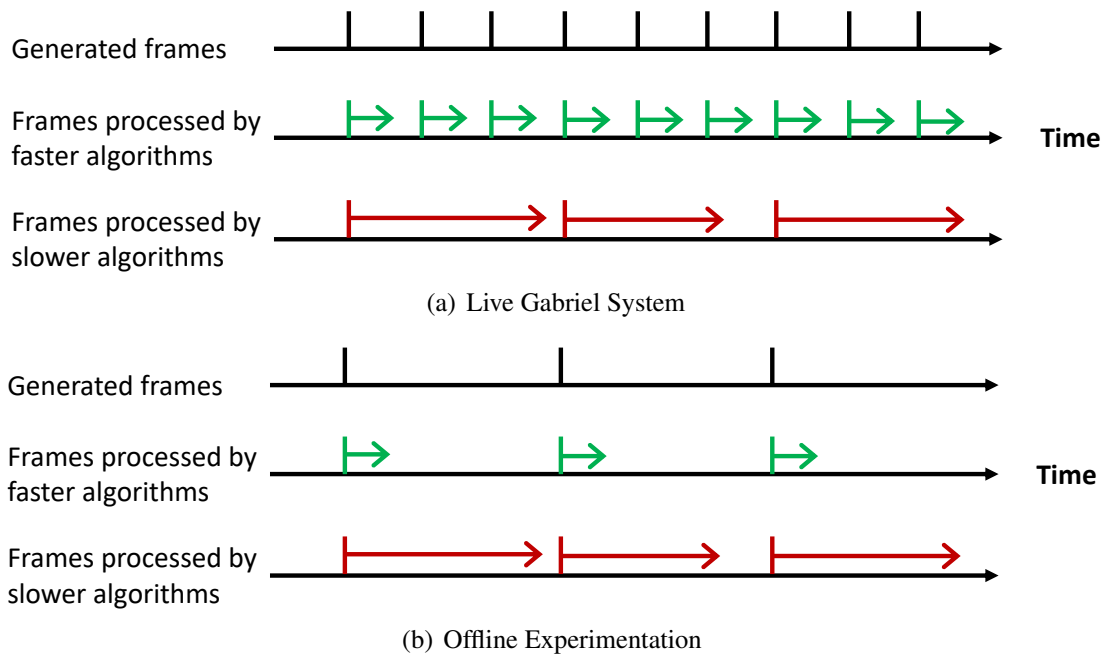


Figure 4.4: Processing Rate for Different Algorithms

Figure 4.3 shows how this approach can be implemented in the Gabriel architecture. Symbolic extractors using different algorithms are run as independent instances in different Cognitive VMs. All of their results are sent to the User Guidance VM, where a result filter decides which result can be trusted. If a result is deemed accurate, it will be used for the following guidance generation step. Otherwise, the result is simply ignored. Note that this architecture is a black box approach, and can be applied even if the source code of algorithms are not available.

In addition to the new result filter component, a small extension to the Gabriel platform is needed. As originally designed, Gabriel implements a pub-sub backbone so that all Cognitive VMs will receive a copy of the sensor stream once they register themselves in the pub-sub backbone. Here, however, there may be multiple identical Cognitive VMs attached to the backbone for higher throughput of one particular cognitive algorithm. In such cases, the Gabriel platform will distribute sensor data (e.g. video frames) to only one idle instance of each algorithm. The Control VM uses the module ID to distinguish different cognitive algorithms, which is provided by each Cognitive VM as they attach to the pub-sub system.

4.2.1 The Basic Algorithm for Result Filter

The operation of the result filter is described by Algorithm 1. Two important data structures are maintained: the *result_history* array tracks recent symbolic representation results from all algorithms, and the *confidence* array counts how many times each algorithm has been consecutively correct. The `process` procedure is called whenever a new result is generated by any of the Cognitive VMs, and received by the User Guidance VM. It then uses the `trust` function to

Algorithm 1 Basic Multi-algorithm Approach (MA1)

```
1: procedure PROCESS(result, algorithm, frame_id)
2:   if TRUST(algorithm, result) then
3:     Feed result to guidance generator
4:   else
5:     Mark result as useless
6:   end if
7:   if algorithm = best_algorithm then
8:     UPDATE_CONFIDENCE(result, frame_id)
9:   else
10:    result_history.add(frame_id, (algorithm, result))
11:  end if
12: end procedure
13:
14: function TRUST(algorithm, result)
15:   if confidence[algorithm]  $\geq$  TH then
16:     return True
17:   else
18:     return False
19:   end if
20: end function
21:
22: procedure UPDATE_CONFIDENCE(best_result, frame_id)
23:   detected_results  $\leftarrow$  result_history.get(frame_id)
24:   for (algorithm, result) in detected_results do
25:     if best_result = result then
26:       confidence[algorithm]  $\leftarrow$  confidence[algorithm] + 1
27:     else
28:       confidence[algorithm]  $\leftarrow$  0
29:     end if
30:   end for
31: end procedure
```

decide whether the result can be used, based on the algorithm’s recent performance. If the result is from the best algorithm we have, we treat its result as the truth, and use it to update confidence scores of the other algorithms through the `update_confidence` procedure.

In this basic implementation, the `update_confidence` procedure simply adds one to the confidence score of an algorithm once it generates a result that matches the result from the best algorithm, and resets the confidence score to zero if it doesn’t. The `trust` function then simply compares the associated confidence entry to a fixed threshold, TH , to decide whether it should be trusted. For example, if the threshold is set to three, then the system will trust the result from a particular algorithm when and only when it matches the result from the best algorithm for the past three times.

Note that the accuracy of fast algorithms can only be checked at speed of the most accurate (and usually slowest) algorithm. Figure 4.4(a) illustrates this: the faster algorithm may be able to keep up with the framerate of the data fed into the system, but the more accurate algorithm is usually much slower and has to skip frames. For example, if there are two algorithms, running at 30 and 1 FPS, then the results will be produced every 33 ms when the fast one is trusted, but the confidence of it can be updated only once a second.

4.2.2 Variants of Result Filter

In practice, the `trust` and `update_confidence` function can be much more complex than the ones shown earlier. In this section, I describe three more variants of these functions. Section 4.3 will compare their performance. For simplicity, I refer to the original multi-algorithm approach shown in Algorithm 1 as MA1, and I will call the three variants MA2, MA3 and MA4, respectively.

The first variant, MA2 (Algorithm 2), has an identical `trust` function that compares the confidence score with a predefined threshold. However, in updating the confidence, it uses a AIMD (additive increase and multiplicative decrease) approach: when the result from an algorithm matches that of the best algorithm, the confidence score increases by one as usual; when it doesn’t, the score drops by half instead of being reset to zero as originally described. This approach prevents an accidental error from ruining the confidence of an algorithm.

MA3 (Algorithm 3) leverages additional output information from an algorithm. For many algorithms, including SVM, the inference result also comes with a probability score, ranging from zero to one, indicating the likelihood the result is correct. It is natural to have more trust on a result with a higher probability score and less trust to that with a lower one. Therefore, MA3 sets a different threshold for algorithm confidence for high probability results (TH_{high}) and low probability results (TH_{low}).

The last variant, MA4 (Algorithm 4), has two improvements over the original form. It keeps a separate confidence score for each category that may be reported by a detection algorithm. This is particularly helpful when an algorithm has different precision for different categories. In addition, it treats “nothing detected” as a special case, and only trusts this result from the best algorithm. This tweak helps reduce false negatives due to fast algorithms that are tuned to have

Algorithm 2 Result Filter for Multi-algorithm Approach: MA2

```
1: The TRUST function is identical to that in Algorithm 1.
2:
3: procedure UPDATE_CONFIDENCE(best_result, frame_id)
4:   detected_results  $\leftarrow$  result_history.get(frame_id)
5:   for (algorithm, result) in detected_results do
6:     if best_result = result then
7:       confidence[algorithm]  $\leftarrow$  min(confidence[algorithm] + 1, TH)
8:     else
9:       confidence[algorithm]  $\leftarrow$   $\lfloor$  confidence[algorithm] / 2  $\rfloor$ 
10:    end if
11:  end for
12: end procedure
```

Algorithm 3 Result Filter for Multi-algorithm Approach: MA3

```
1: function TRUST(algorithm, result, probability)
2:   if probability  $\geq$  mean_probability and confidence[algorithm]  $\geq$  TH_low or probability  $\leq$ 
   mean_probability and confidence[algorithm]  $\geq$  TH_high then
3:     return True
4:   else
5:     return False
6:   end if
7: end function
8:
9: The UPDATE_CONFIDENCE function is identical to that in Algorithm 1.
```

Algorithm 4 Result Filter for Multi-algorithm Approach: MA4

```
1: function TRUST(algorithm, result)
2:   if confidence[algorithm, result]  $\geq$  TH then
3:     return True
4:   else
5:     return False
6:   end if
7: end function
8:
9: procedure UPDATE_CONFIDENCE(best_result, frame_id)
10:  detected_results  $\leftarrow$  result_history.get(frame_id)
11:  for (algorithm, result) in detected_results do
12:    if best_result = result and result  $\neq$  None then
13:      confidence[algorithm, result]  $\leftarrow$  confidence[algorithm, result] + 1
14:    else
15:      confidence[algorithm, result]  $\leftarrow$  0
16:    end if
17:  end for
18: end procedure
```

high precision but low recall.

4.3 Evaluation: an Offline Analysis

In this section, I will conduct an in-depth study of the multi-algorithm approach using Face as an example application. I will show seven different candidate algorithms that I have implemented for Face, which possess different speed and accuracy tradeoffs. I will then investigate how the algorithm performance is affected by different variants of result filter algorithms as well as the threshold parameter. Lastly, I will demonstrate proofs that the temporal locality of algorithm accuracy is the key to make this approach work.

Specifically, I explore answers to the following questions:

- Can the multi-algorithm approach result in a better speed-accuracy tradeoff?
- Is having more algorithms helpful?
- Which variant of the result filter gives the best result?
- How does the threshold parameter in the result filter affect performance?
- Will the multi-algorithm approach work with randomly shuffled video frames?

4.3.1 Experimental Approach

In an experiment with the Gabriel system, every run will generate different latency and accuracy measurement results even if it is fed with the same dataset, due to the variations of network transmission time, processing time, and different frames being skipped by different algorithms. These differences, although usually small, may overshadow the differences caused by different mechanisms or parameters in result filters. In order to look more clearly at the effect of algorithm configurations, I use an offline analysis approach in this section.

I first run every candidate algorithm for Face through the whole dataset and log its processing result and latency into a file. Then, in all later experiments, the processing result and latency of each algorithm is read from these files so that they are identical through different runs. In this way, I can focus on studying the effect of different algorithm selection and parameter settings. Since I also want to remove the randomness caused by skipping different frames, I let all algorithms process all frames so that the algorithm switching can happen at the frame generation rate. This is effectively like feeding frames into the system in a very low rate, as shown in Figure 4.4(b). As I will show later in Section 4.4, this change of algorithm switching rate has little impact on the overall latency and accuracy.

I have implemented seven different algorithms for the Face application, as summarized in Table 4.1. The first algorithm, A1, is the same algorithm used in my experiments in Chapter 3 that uses a HOG-based face detector with a DNN [80] for accurate face recognition. A2, A4, and A6 use the same face detector, but use Fisherface [29], Eigenface [171], and Local Binary Patterns Histograms (LBPH) [22] for face recognition, respectively. The rest three algorithms,

	Face detection	Face recognition	Precision (%)	Recall (%)	Average latency (ms)
A1	HOG-based	DNN	99.4	96.6	230.8
A2	HOG-based	Fisherface	80.7	78.0	133.6
A3	Haar-cascade	Fisherface	70.2	67.9	47.8
A4	HOG-based	Eigenface	63.5	61.4	137.8
A5	Haar-cascade	Eigenface	52.8	51.1	50.7
A6	HOG-based	LBPH	32.9	31.9	183.7
A7	Haar-cascade	LBPH	19.0	18.4	97.5

The latency measurements are done on a Dell Optiplex 9020 machine with four Intel® Core™ i7-3770 CPU cores. Note that latency here means only the processing time. It does not include network transmission delay.

Table 4.1: Candidate Algorithms for Face

in addition, use Haar-cascade face detector for faster detection [175].

Obviously, the HOG+DNN based algorithm (A1) has the highest accuracy, but is also the slowest. Fisherface (A2) is the second most accurate, and also faster. Switching to Haar-based face detector (A3) further increases speed and reduces accuracy. These three algorithms form a nice speed-accuracy tradeoff where no one outperforms another in both dimensions. Most of the experiments, therefore, will be using these three algorithms only. The rest algorithms (A4, A5, A6, A7) are not as performant as the previous ones, but I'll also show the impact of adding them to the algorithm candidate pool of the system.

4.3.2 Performance of Multi-algorithm Approach

Table 4.2 shows the new accuracy and latency values achieved by the multi-algorithm approach. The first three lines are numbers copied from Table 4.1 for easier comparison. As we can see, using the basic result filter algorithm (MA1), the new approach achieves a pretty good speed-accuracy tradeoff. Comparing with the second best algorithm (A2), it is now both more accurate and faster. Using the AIMD approach in MA2 improves the overall accuracy, at the cost of higher latency. This difference is very small, and is mostly because of the different threshold value settings, which will be illustrated in Section 4.3.4.

Similarly, MA3 has little impact on the latency and accuracy of the combined approach. This is surprising, since one might believe that the probability score must be good hints to the quality of a recognition result. However, as Figure 4.5 shows, there is no strong correlation between probability score and its correctness. In this figure, each scattered point represents whether the result is correct, and the probability score associated with that result in algorithm A3. Obviously, there are many low probability results that are correct, and high probability ones that are incorrect, which proves the weak correlation between the two attributes.

Finally, MA4 is also slower than MA1, but slightly faster than both MA2 and MA3. At the same time, it achieves a much higher accuracy – almost as accurate as the best algorithm that

	Precision (%)	Recall (%)	Accuracy (%)	Average latency (ms)
A1	99.4	96.6	97.4	230.8
A2	80.7	78.0	85.5	133.6
A3	70.2	67.9	76.7	45.8
A1 + A2 + A3 (MA1)	82.0	81.1	85.9	79.6
A1 + A2 + A3 (MA2)	83.6	82.8	87.1	87.9
A1 + A2 + A3 (MA3)	83.0	82.2	86.9	86.1
A1 + A2 + A3 (MA4)	98.4	96.3	96.9	81.9
A1 + A3 (MA4)	98.8	96.7	97.2	103.7
A1~A7 (MA4)	97.8	96.3	96.8	66.7

Refer to Table 4.1 for the description of A1 to A7. MA1, MA2, MA3, and MA4 are variants of result filters described in Section 4.2. “A1~A7” means A1 through A7. In all multi-algorithm approaches, the threshold is set to 2.

Table 4.2: Candidate Algorithms for Face

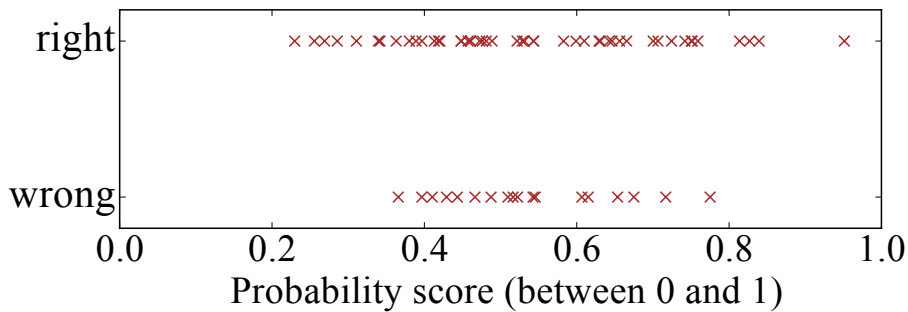


Figure 4.5: Correlation between Result Probability and Correctness

uses DNNs. This is because keeping different confidence score for different categories helps to leverage good results from the faster algorithm even when its general accuracy is low. For example, a face recognition algorithm may confuse between Bob and John when it sees the face of Bob, resulting half of Bob’s faces incorrectly reported as John. With MA4, the system will set a high confidence score when it reports “Bob” and low score for “John” in a certain period of time, thus speeding up half of the frame processing while preserving high accuracy.

The speed-accuracy relationship of different techniques can be easily compared in Figure 4.6(a), where the more up left, the better accuracy and speed. The three red dots of different darkness represent individual algorithms, while the green dots show results using different multi-algorithm approaches. Clearly, all three are an improvement over A2. And the approach with MA4 looks almost like shifting A1 directly to the left, which means it reduces nearly half of the latency with little impact on accuracy. In general, using the MA4 algorithm for result filter leads to significantly higher accuracy than MA1, MA2, and MA3, while achieving decent speed. Therefore, I will use MA4 for the rest of the experiments.

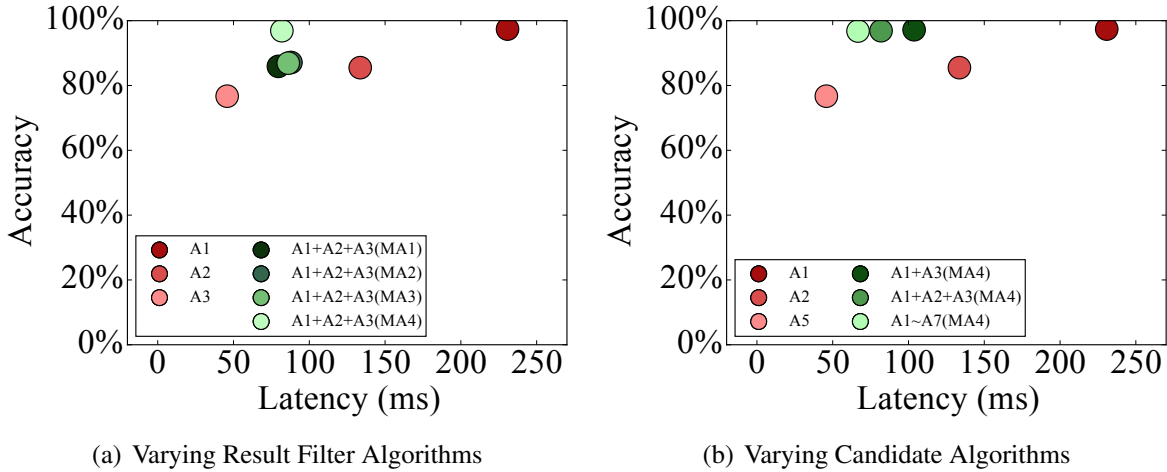


Figure 4.6: Speed-accuracy Tradeoff of Different Algorithms in Face

4.3.3 Varying Number of Candidate Algorithms

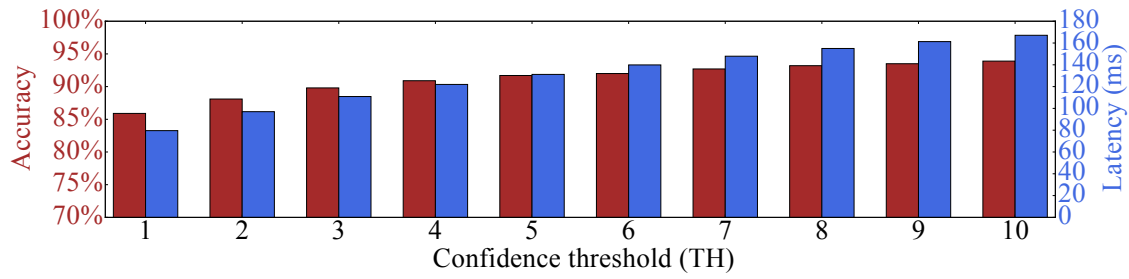
Next, I vary the number of candidate algorithms being used and see how this affects the overall latency and accuracy. I first did an experiment with only two candidate algorithms: the most accurate (A1) and the fastest (A3). I then experiment with all seven algorithms A1~A7. Table 4.2 and Figure 4.6(b) show the new speed-accuracy tradeoff achieved with these approaches.

Clearly, as we have more candidate algorithms, the chances that some of the faster algorithms have enough confidence score increases so that less frames have to wait until the slowest algorithm to finish. This gives some improvement in speed, which comes at the cost of slight accuracy drop. In general, varying the number of candidate algorithms does not change the overall result very much. On the other hand, the hardware resource needed for concurrently running such candidate algorithms grows linearly to the number of candidate algorithms. Therefore, in the following experiments, I will use three algorithms, which won't require too much hardware but are still able to demonstrate the functionality and effectiveness of multiple algorithms.

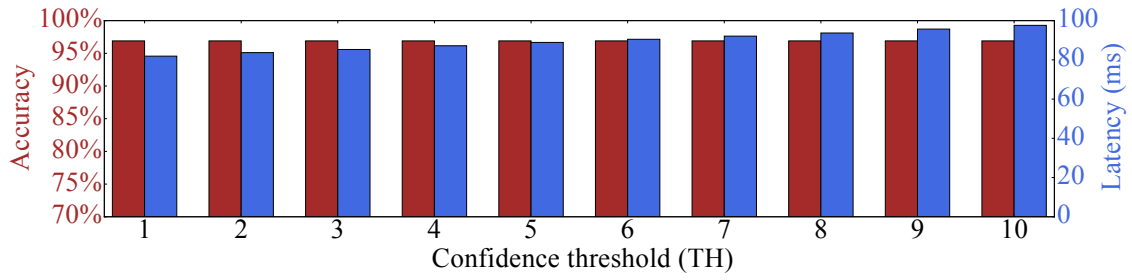
4.3.4 Effect of Threshold Value

So far I have set the confidence threshold to the lowest number possible. For MA1 and MA4, this was one. In MA2, this was set to two since it is the smallest number to show the effect of an AIMD approach. In MA3, TH_{low} and TH_{high} were one and two, respectively. This section explores how much does the threshold value affect system performance.

I vary the threshold from one to ten using A1, A2, and A3 as candidate algorithms, and experiment with the multi-algorithm approach MA1 and MA4. Figure 4.7 plots the average latency and accuracy as the threshold varies. As one would expect, when the threshold increases, both accuracy and latency increase. This is because a higher threshold value reduces the chance of a faster and less accurate algorithm being trusted. For MA1, the accuracy increases from



(a) MA1



(b) MA4

Figure 4.7: Effect of Confidence Threshold (TH) for Face

85.9% when TH equals to one, to 91.7% when TH is five. The improvement speed slows down as TH becomes bigger, and the accuracy finally reaches 93.9% when TH is ten. The latency increase follows a similar pattern, from 79.6 ms ($TH=1$) to 167.1 ms ($TH=10$). When using MA4, the increase of latency is almost linear to the threshold increase, while the accuracy level remains almost the same. Therefore, to achieve the highest accuracy at the lowest latency, I set threshold to be one for MA4 in the following experiments.

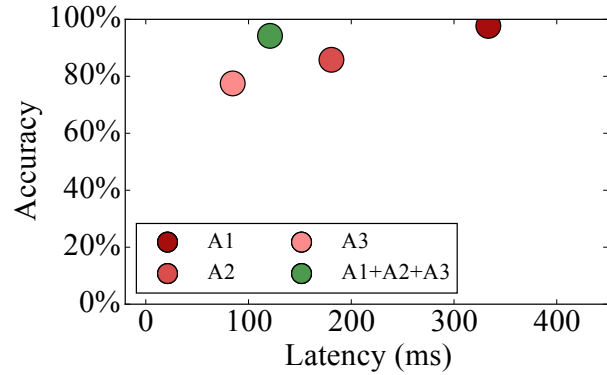
4.3.5 Randomizing input image order

As mentioned in Section 4.1, temporal locality is a key to opportunistically using results from faster algorithms while still maintaining high accuracy. How important is this temporal locality for the multi-algorithm approach? To quantify this, I did an experiment with the same dataset, but with randomized input frame order so that temporal locality no longer exists.

As a result, with MA4 (and A1, A2, and A3 as candidate algorithms), the average latency remains similar to before (about 90 ms), but the average accuracy is less than 20%. This is due to the fact that even if a faster algorithm has been correct for several consecutive frames, trusting it for the next frame is still a random guess since temporal locality is lost. Therefore, this shows a strong proof that temporal locality is the key to the success of the multi-algorithm approach.

Algorithms	Accuracy (%)	Latency (ms)
A1: DNN+HOG	97.7	333.4
A2: Fisher+HOG	85.8	180.7
A3: Fisher+Haar	77.5	84.6
Multi-algo	94.2	120.9

(a) Accuracy and Latency

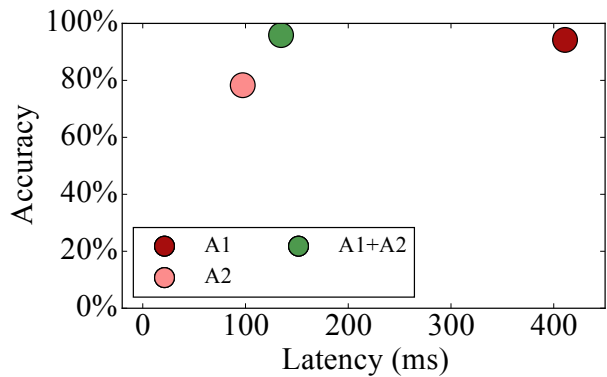


(b) Tradeoff Plot

Figure 4.8: Multi-algorithm Approach Result for Face

Algorithms	Accuracy (%)	Latency (ms)
A1: Lego-robust	94.2	411.0
A2: Lego-simple	78.3	97.3
Multi-algo	95.9	134.6

(a) Accuracy and Latency

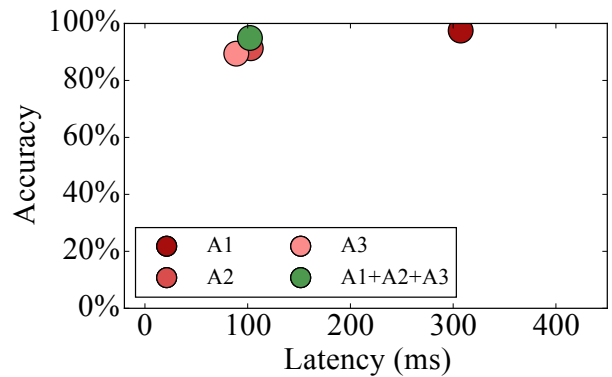


(b) Tradeoff Plot

Figure 4.9: Multi-algorithm Approach Result for Lego

Algorithms	Accuracy (%)	Latency (ms)
A1: DNN1	97.5	307.3
A2: DNN2	91.3	103.0
A3: DNN3	89.4	88.9
Multi-algo	94.9	102.2

(a) Accuracy and Latency



(b) Tradeoff Plot

Figure 4.10: Multi-algorithm Approach Result for Sandwich

Algorithm	Detection Result Sequence (P: positive, N: negative)												Accuracy	
Ground truth	P	P	P	P	P	P	P	P	P	P	N	N	N	
Algo-fast (less accurate)	N	N	P	N	P	P	P	P	P	N	N	N	N	69.2%
Algo-slow (more accurate)	P	P	P	P	P	N	P	P	P	P	N	N	N	92.3%
Multi-algo	P	P	P	P	P	P	P	P	P	P	N	N	N	100.0%

Table 4.3: Example Detection Sequence where Multi-algorithm Approach Has the Best Accuracy

4.4 Evaluation on Gabriel

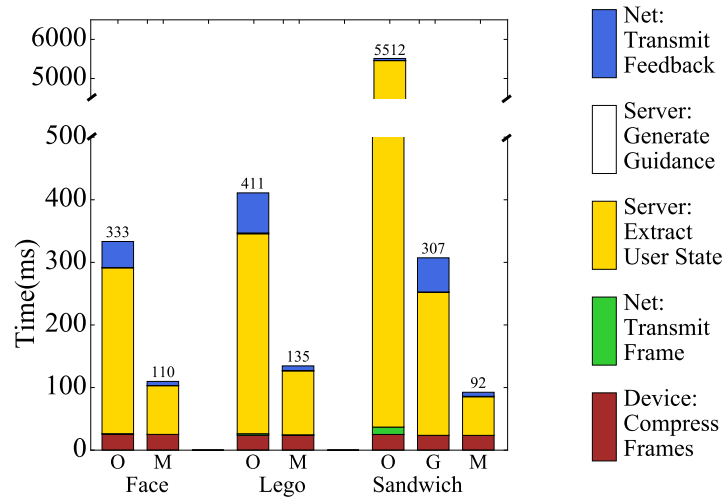
This section demonstrates the multi-algorithm approach applied to three of my applications running on top of the Gabriel platform. The first is the Face application used in the previous section. Since now frames are generated at 15 FPS, the confidence score of each algorithm can only be adjusted every few frames. I’ll show how this affects the overall accuracy.

The other two applications are Lego and Sandwich. For Lego, the original algorithm spends much computation compensating for lighting variations and blurriness. Therefore, I implemented a faster Lego-simple algorithm that removes much of this complexity, but may fail to detect the Lego shape when lighting isn’t ideal. For Sandwich, I implemented three different object detection algorithms using different DNN architectures. DNN1 is based on the VGG16 model [158], and can be considered state-of-the-art in terms of accuracy. DNN2 uses the ZF model [183], while DNN3 implements the VGG_CNN_F model [39].

Figure 4.8, 4.9, 4.10 show the accuracy and latency for each of these algorithms as well as the combined multi-algorithm approach. The left part of each figure shows the absolute numbers, while the right part depicts the speed-accuracy tradeoff in the 2D space. Clearly, the multi-algorithm approach can produce results with very low latency, but with little, if any, sacrifice in accuracy. Across the board, the new approach is consistently much faster than the most accurate algorithm, yet still provides significant accuracy gains over the second best algorithm.

Interestingly, the multi-algorithm Lego even results in slightly higher accuracy than the best algorithm. This is due to the few, rare cases in which the faster algorithm is trusted and matches the ground truth, but the slower algorithm makes a mistake. Table 4.3 shows an example of such cases. In this example object detection task, the slow algorithm achieves higher accuracy than the faster one overall. However, it may occasionally make a mistake, as shown in the “P N P” sequence marked in red. For the frame it reports “N”, the faster algorithm produces correct result, and will be used, since its result for the previous frame matches that of the slow algorithm. Thus, the mistake is hidden in the multi-algorithm approach, resulting in higher accuracy overall.

Figure 4.11 shows the time breakdown for the multi-algorithm approach. Not surprisingly, most of the gains are due to faster symbolic representation extraction at the server. However, result transmission is also improved. This is because with the faster approach, the system operates at a higher frame rate, so device radios do not enter deep, low-power states. Thus, latency overheads of power-state transitions of the wireless network interfaces are avoided. Overall, my multi-algorithm approach demonstrates that by exploiting temporal locality, one can combine algorithms to get the best of both accuracy and speed. Furthermore, this technique shows a way



O: Original Algorithm G: Original Algorithm (w/ GPU)
M: The Multi-algorithm Approach

Figure 4.11: Multi-algorithm Latency Breakdown

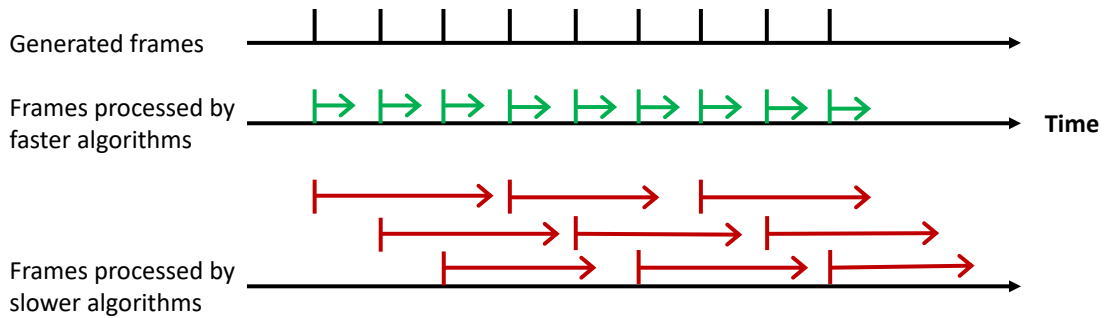


Figure 4.12: Processing Rate for Different Algorithms with Multiple Best-algorithm Instances

to use additional processing resources to improve latency, not just throughput, of a processing task, without deep change to its algorithms.

Effect of Confidence Update Frequency

As pointed out in Figure 4.4, the confidence update of the faster algorithms can only happen at the processing rate of the best algorithm, which is different from the offline experimentation setup, where the accuracy is checked at full frame rate. However, in comparing Figure 4.8 with Table 4.2, we can see that such difference does not result in significant accuracy drop for the multi-algorithm approach. Note that the latency numbers in Figure 4.8 and Table 4.2 are not directly comparable, as the results in Table 4.2 did not include networking time.

One approach to further improve system accuracy is to run multiple instances of the best algorithm concurrently (Figure 4.12). This allows for the algorithm confidence to be updated

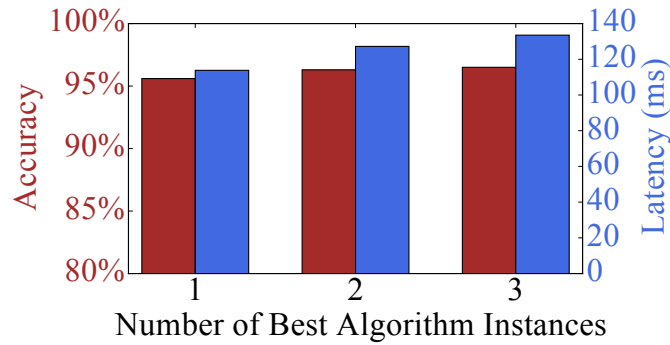


Figure 4.13: Effect of Having Multiple Best-algorithm Instances (Face)

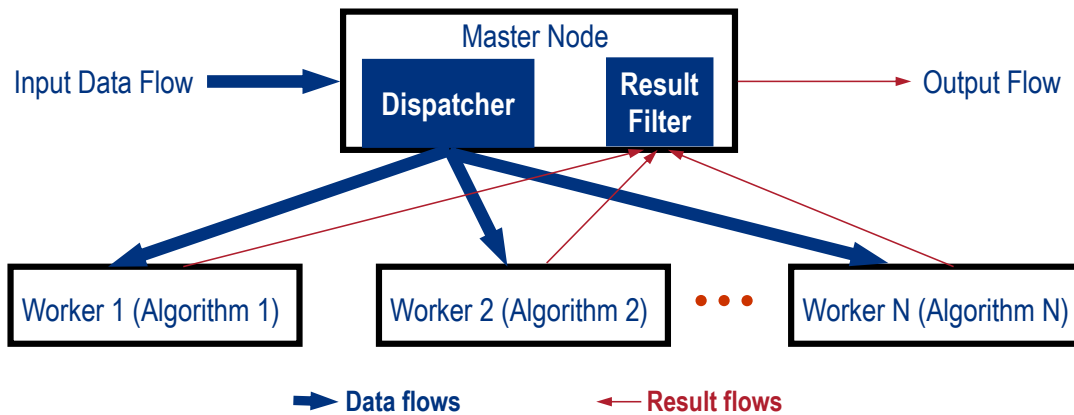


Figure 4.14: General System Workflow to Leverage Multi-algorithm Approach

at a higher rate, although there is still a delay in the update. For example, if we again assume two candidate algorithms each processing at 1 FPS and 30 FPS, when there are sufficient best-algorithm instances, the confidence of the faster algorithm can be adjusted at 30 FPS, but with a delay of about one second. Figure 4.13 shows the experimental results of having multiple best-algorithm instances for the Face application. Clearly, with more instances, the multi-algorithm approach achieves slightly higher accuracy, but with higher latency as well. In general, the multi-algorithm approach is able to achieve very high accuracy at fast speed, even with the delay and low frequency of confidence update in a live system.

4.5 Applying Multi-algorithm Approach on Other Systems

Although originally developed to speed up wearable cognitive assistance applications running on Gabriel, the multi-algorithm approach can be applied to many other systems to speed up a specific task. To use this approach, two requirements need to be met. First, for the same task, there must be more than one algorithm that achieves different speed-accuracy tradeoff. Second, there must be sufficient hardware resources to run these algorithms concurrently at run time.

Figure 4.14 shows a general system that leverages this approach. In this architecture, a master

Algorithms	Video 1		Video 2		Video 3	
	Accuracy (%)	Time (ms)	Accuracy (%)	Time (ms)	Accuracy (%)	Time (ms)
A1: TLD	70	11567	82	12785	91	99312
A2: MF	31	1449	43	639	61	3243
A3: KCF	5	1122	28	866	54	8112
Multi-algo	85	2228	85	1623	92	7951

Video 1: pedestrian2 Video 2: pedestrian3 Video 3: car
All three videos are from the TLD dataset [100].

Table 4.4: Multi-algorithm Approach Result for Object Tracking

node takes input data from network (e.g. in Gabriel) or from disk (e.g. process a pre-recorded video file), and dispatches the data units to idle worker nodes that run different algorithms. Some of the worker nodes may be running the same algorithm to purely improve throughput, similar to the approach in Section 4.4. All results from the worker nodes are then sent back to the master, where the previously shown result filter module will be used to decide which ones to trust. Finally, all of the results that could pass the check of the result filter are sent to the output, either back to the network, or written to the disk.

In this section, I demonstrate the use of such a system using object tracking in a video file as an example. Fast object tracking can be very helpful to quickly label training data for object detection. For example, in TPOD [178], a web-based training framework, a user is only required to manually draw object bounding boxes in the first few frames in a short video, and the remaining frames can be labeled automatically using tracking. The user then occasionally adjusts the bounding box’s position when tracking is off. However, accurate tracking can be slow, so this may still consume a lot of user time. This could be significantly improved by the multi-algorithm approach.

I have implemented three different tracking algorithms. The first and most accurate one is using the tracking-learning-detection (TLD) approach [100]. It continuously revises an object detection model while tracking the frames, and applies it to new frames to calibrate tracking error. Therefore, it is able to re-detect an object even after losing track of it. The second algorithm uses median flow (MF) [99]. It is the best when the object has no occlusion and moves smoothly. Finally, the fastest algorithm is based on kernalized correlation filters (KCF) [47, 81].

One thing that differentiates this example from the applications shown in the previous section is that trackers are stateful. Each tracker leverages object location from previous frames to track the next frame. In my implementation of the multi-algorithm system, therefore, once a result from the faster algorithm does not match that from TLD, it is deemed to have lost track. The master node will then send a tracker re-initialization command to the corresponding worker node, using the result from TLD as the initial object position.

I test these algorithms, as well as the multi-algorithm approach, using three video sequences from the TLD dataset [100]. Their accuracy and the total time to track the whole video are summarized in Table 4.4, and plotted in Figure 4.15. Overall, the multi-algorithm approach achieves a great tradeoff between accuracy and speed. The MF and KCF algorithms have very poor ac-

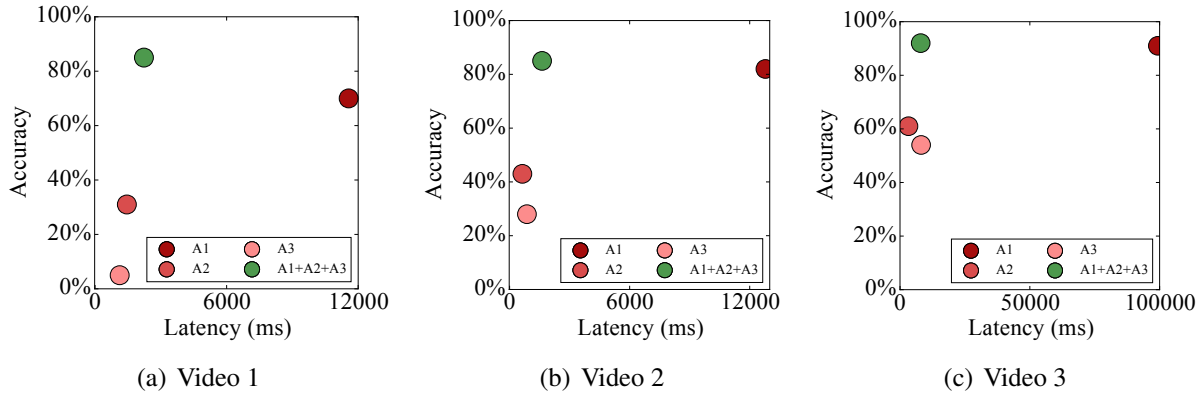


Figure 4.15: Speed-accuracy Tradeoff of Different Algorithms for Object Tracking

curacy on their own, but they help to achieve a higher accuracy in the multi-algorithm approach than the best individual algorithm. This is partially due to that once tracking of MF or KCF is lost, they are not able to recover by themselves. However, with the help of re-initialization command from the master node when running them concurrently, these algorithms can generate more accurate results than they would have when running independently. In terms of time consumption, the multi-algorithm approach is 5x to 15x faster than TLD, which will save a user huge amount of time in a system like TPOD.

4.6 Discussion

4.6.1 An Analogy

To me, it has been helpful to think about the multi-algorithm approach through an analogy to the college study experience. When a student is studying the textbook and is confused with one section, she can choose to email the instructor (similar to the best algorithm), or one of the TAs (similar to the other candidate algorithms) for clarification. The instructor’s explanations are usually the most trustworthy, but the responses may not be very prompt since the instructor is busy dealing with thousands of emails everyday. The TAs, on the other hand, can be more responsive, but their explanations may also consist of flaws. My multi-algorithm approach is like sending emails to all of them at the same time. Once the student receives the first answer that she believes is good enough, she moves on to study the next section.

In this process, the student can build up a confidence score table for all the TAs. The simplest way to define confidence is something like “Was his answer consistent with the instructor’s answer last time?”, or “How about the time before last?” If one of the TAs has been consecutively giving high-quality answers, then it is reasonable to rely on its answer the next time. This is exactly what I did in MA1. Of course, the TAs may be good at some topics but not the others, so it is plausible to build the confidence score per topic, and start to ask questions such as “How did this TA perform when he last answered questions regarding Paxos?” Now this meta-algorithm

has become very similar to MA4.

In reality, one may not depend on the explanation to the previous section to study the following material. In fact, the understanding of different sections may not be relevant at all. In such cases, it may be harder to judge a TA's answer to one question based on his performance in the previous one. Therefore, the temporal locality of accuracy is really important to the success of such approaches.

4.6.2 Limitations

The current approach possesses several limitations. First of all, the outcome of combining multiple algorithms may still not be the best to use. Although it usually achieves a nice tradeoff between speed and accuracy, it is usually less accurate than the best candidate algorithm, and slower than the fastest candidate algorithm. Therefore, if a specific application values either accuracy or speed greatly more than the other, the original individual algorithm that optimizes for either accuracy or speed should still be used.

Second, the multi-algorithm approach results in lower latency on average, but at the same increases its variation. In other words, in a wearable cognitive assistance application, a user may receive fast responses for several minutes (or seconds) but encounter more delays for the next minute (or second). This jitter may negatively impact a user's experience in some applications [140, 180]. For applications such as object tracking shown in Section 4.5, this will not be a problem, since the only time that matters is the total completion time.

Third, all applications I have experimented with this approach apply processing on individual frames. The frame-level temporal locality of accuracy has been proven, but it is unclear whether consecutive video chunks exhibit the same behavior. For example, if the task is to detect human action from a short period of video, can the multi-algorithm approach still help? This is a question to be answered in the future work.

Finally, running multiple algorithms concurrently means provisioning more hardware resource to the same user. In a cloud or cloudlet that serves multiple users, this may not be possible at peak time, even if the user is willing to pay more. How to assign compute resource to different users so that most people are satisfied with the applications is a much more complex scheduling and optimization problem. This is out of the scope of this dissertation, which focuses on studying and optimizing the performance of a single user. Therefore, I will leave the problem of multi-user resource scheduling to future work as well.

Chapter 5

Optimizing Energy Consumption for Wearable Cognitive Assistance

Battery life has always been a major design consideration in building mobile applications. This is especially true for applications targeting wearable devices, whose battery capacity is constrained by their weight, size, and form factor. Currently, the Gabriel client software can only last about 50 minutes on a Google Glass. This number is not going to improve much simply with hardware upgrade, since battery technology does not follow Moores Law. Therefore, this chapter explores different approaches to conserving energy and extending battery life for wearable cognitive assistance applications.

Recent research efforts have suggested that sensor reading (e.g. GPS, camera), computation (e.g. CPU), and network transmission are among the biggest energy consumers for mobile applications [115, 154]. Thus, an effective way of conserving energy is to get rid of unnecessary sensor reading, processing, and data transmission of an application. For example, for applications such as Lego and Sandwich that provide step-by-step instructions, the camera can be turned off while the user is performing assembly and hasn't finished the current step. The idea of using cheap sensors to trigger more expensive sensors can also help, as studied in [179]. All of these approaches require fine-grained sensor control policies targeting a specific application. Fortunately, as mentioned in Section 2.3.5, the Gabriel platform provides a simple interface to allow such fine-grained control. In Section 5.1, I will first profile the power consumption of different sensor components of a device, and quantify the potential benefits of sensor control. I will then use Ping-pong and Lego as two example applications to demonstrate the usage and effectiveness of such approaches in Section 5.2 and Section 5.3, respectively.

Besides application specific optimization, modern mobile devices have implemented system-wide approaches to conserving energy. For example, many devices support power saving mode (PSM) that tries to put radio hardware into sleep mode as much as possible. However, this usually comes at the cost of additional delay of network transmission, thus hurting the responsiveness of an application. Section 5.4 studies the tradeoff of enabling PSM on example cognitive assistance applications, and offers suggestions to achieve optimal speed-energy balance.

(a) Nexus 6 Phone

Sensor States			Mean Power (W)	Mean Current (mA)	Battery Life (h)
Image	Audio	Accelerometer			
off	off	off	1.1	262	12.3
on	off	off	2.9	728	4.4
off	on	off	1.7	413	7.8
off	off	on	1.6	386	8.4
on	on	on	2.9	749	4.3

(b) Google Glass

Sensor States			Mean Power (W)	Mean Current (mA)	Battery Life (h)
Image	Audio	Accelerometer			
off	off	off	1.0	272	2.09
on	off	off	2.1	636	0.90
off	on	off	1.3	343	1.66
off	off	on	1.5	411	1.39
on	on	on	2.2	655	0.87

Table 5.1: Power Consumption of Different Sensor States

5.1 Profiling Power Consumption of Gabriel Applications

This section profiles the power consumption of different sensor readings on the mobile device, as well as the associated encoding and network transmission of sensed data. The power difference between turning on and off a sensor shows the potential of power savings we can get if we optimally control the usage of a particular sensor. The comparison among different sensor power consumption leads to the focus of further optimization.

I measured the power consumption of the Gabriel client by switching on and off each of the sensors currently supported by Gabriel, namely the camera, the microphone, and the accelerometer. The server back-end is running the dummy application as described in Section 2.4.2, which simply echos every received message. After several runs of a five-minute experiment, Table 5.1 summarizes the measured average power, average current, and associated battery life for two devices, the Nexus 6 Phone, and the Google Glass. The battery life was calculated by dividing the battery capacity by average current.

Clearly, camera is the most power hungry sensor on both devices. Purely turning on the camera and streaming images can cut the battery life by more than a half on both devices. In contrast, the other two sensors are relatively cheaper, as they only reduce battery life by 32% to 37% on the phone, and 21% to 33% on the Google Glass, with network being active all the time. In addition, while the images are already being captured and streamed, turning on the cheap sensors add minimal overhead. This is because their sensing functionality consumes little power compared to the camera, and the added network traffic has little impact on energy since the radio is already active.

These results suggest a large potential in saving energy if we carefully control these sensors. Ideally, given the relative ratio of power consumption between “idle” state and “camera on” state, if the system is clever enough to only turn on the camera to capture a few frames at the critical moments, while keeping the device idle most of the time, the battery life can be more than doubled. More practically, even if the camera usage can only be reduced by half, the battery life can still be improved by more than a quarter. In addition, the significant power difference of different sensors confirm the potential benefit of using cheap sensors to continuously monitor a user’s state and only trigger the use of a camera when necessary.

Note that another big energy consumer of mobile devices is the screen. Conserving this part of energy will not be studied in this chapter, as either turning off or dimming the screen will negatively affect a user’s perception of the visual instructions (e.g. image or animation). Ideally, the screen can be controlled in a fine-grained way such that it is displaying with full brightness when the user is looking at the screen, but darkened as soon as the user turns attention to real-world objects. However, this requires accurate tracking of the user’s gaze, and is left as part of the future work.

5.2 Application-specific Sensor Control: Ping-pong

Many opportunities lie in leveraging application specific characteristics to reduce unnecessary processing, sensing, and network transmission for energy conservation. As described in Section 2.3.5, the Gabriel platform provides a simple interface for fine-grained sensor control of the mobile device. More specifically, for each application running in the Cognitive VMs or the User Guidance VM, sensor control commands can be embedded in the return message through the `control` field, which will be delivered to the mobile device through the control channel. Generally, the control commands can be one of the three types:

- Immediately turn on/off a sensor.
- Turn on/off a sensor with some delay.
- Adjust the resolution of a sensor.

In this section and the one that follows, I will demonstrate how these sensor control mechanisms can be used to reduce energy consumption using two case studies. This section studies Ping-pong, and the next section studies Lego. These sections will show that the application-specific sensor control policies can significantly reduce power consumption of wearable cognitive assistance applications. More importantly, the Gabriel platform has allowed a very simple interface to employ these policies, which simplifies the development of new applications.

5.2.1 Methodology

In the original implementation of Ping-pong, the client device continuously streams image data to the cloudlet for analysis of the ping-pong rally, including the detection of table contour, ball position, and opponent position. This ensures crisp response provided to the user, but also leaves

the camera and radio circuitry always powered on, consuming a lot of energy. In reality, the system can be “lazy” at times. For example, in between two rallies, when the user is walking to pick up the ball or chatting with her friends, the application doesn’t need to be “alert” all the time to provide guidance. In other words, only during each ping-pong rally does the system have to keep the camera on and provide quick response.

Built on this intuition, I implemented a simple sensor control policy illustrated as below.

- Always have the microphone on and continuously process audio data to detect ping-pong sound (algorithm described in the next section).
- Turn on camera as soon as ping-pong sound is detected.
- Turn off camera if ping-pong sound is not detected within 2 seconds.

Since audio sensing is much more power efficient than image sensing, this policy will likely extend the battery life when running Ping-pong, especially when there are long breaks between ping-pong rallies. In practice, the system can be configured differently regarding where the audio processing is done. In most Gabriel-based applications, all sensor streams are transmitted to the cloudlet for complex processing. However, in this case, as shown in the following section, the audio processing needed is relatively simple, thus I also tried to do the processing locally on the mobile device. Section 5.2.4 will compare the performance of these two system configurations.

From the application developer’s perspective, enforcing such a policy only requires embedding the “turn camera on” and “turn camera off” commands based on the audio processing results. All the details of hardware and network management on the device will be automatically taken care of by the underlying Gabriel platform.

5.2.2 Ping-pong Sound Detection

The audio analysis for ping-pong sound detection follows the research work described in [184], which was originally designed for highlight detection for sports games. The detection pipeline consists of two steps. The first step is called Energy Peak Detection. It calculates the Short-time Energy (STE) [185] for every audio frame of 10 ms, which is the typical duration of a ball hit. If this energy is considerably larger than the average STE of the surrounding macro frame (100 ms), the frame likely includes a ball hit. The final decision is then made for every one to two seconds window, based on how many such energy peaks are within the time period.

While the first approach is simple and fast, it may confuse between ping-pong sound and other impulsive background noise, such as tapping and chatting. The second step aims to fix this by applying MFCC-based classification approach [59, 186]. MFCC is one of the most popular spectrum-based audio features in speech recognition and characterizing general audio signals. Once this feature is computed for every 10 ms frame, it can then be fed into a SVM to distinguish between ping-pong sound and background noise. The detailed SVM training process is explained in [184].

(a) Nexus 6 Phone

System Configuration	Power (W)	Current (mA)	Battery Life (h)
Always transmit video	2.9	739	4.4
Audio analysis on cloudlet	2.2	567	5.7
Audio analysis on device	1.9	487	6.6

(b) Google Glass

System Configuration	Power (W)	Current (mA)	Battery Life (h)
Always transmit video	2.1	608	0.94
Audio analysis on cloudlet	1.8	492	1.16
Audio analysis on device	1.6	463	1.22

Table 5.2: Power Consumption of Ping-pong with and without Sensor Control

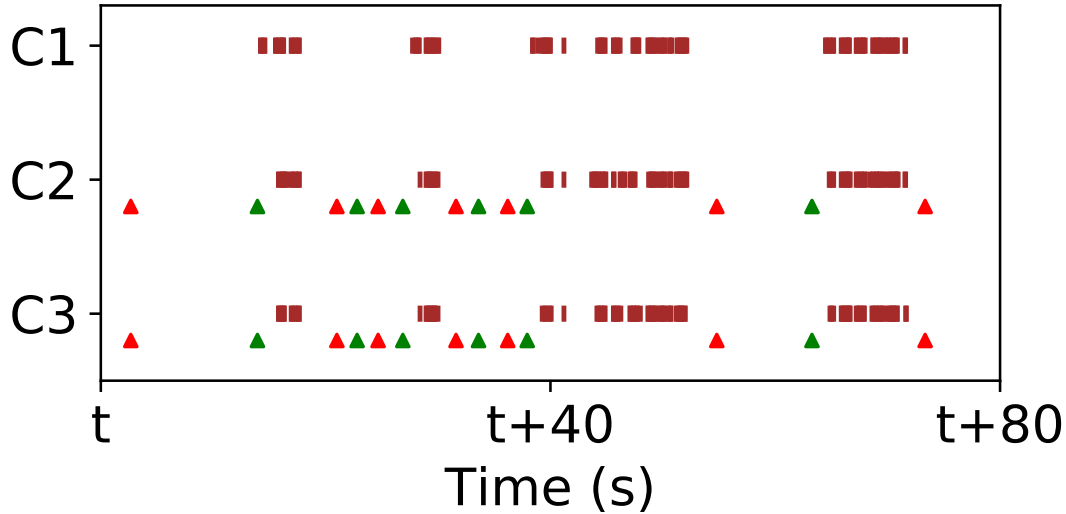
In practice, running MFCC feature extraction is too heavy weight on wearable devices [131]. Therefore, I am only using Energy Peak Detection in my experiments no matter whether the ping-pong sound detection is offloaded to the cloudlet or done natively on the device. In my experience, the second step of MFCC-based classification makes negligible difference to the final sound detection results in the video I collected. It is also important to notice that using such algorithms is purely an example to demonstrate sensor control capabilities of Gabriel. The accuracies of such algorithms are not the focus of my work, and will not be evaluated.

5.2.3 Experimental Setup

Most of the experimental setup is the same as that described in Section 3.3.1. However, different from all previous experiments, to test the effectiveness of sensor control requires more than one sensor streams to be read and transmitted concurrently. When the application is running live, such concurrency won't cause synchronization problems since any data read from a sensor is the latest. However, since I need to use pre-recorded sensor streams in an experiment for reproducibility, time synchronization among different sensor streams require careful implementation.

In my implementation, the video stream is stored and loaded as JPEG images as before (Section 3.3.1). These frames can be time-stamped so that we are able to know exactly their position within a video. The audio stream is extracted from the same pre-recorded video file, but stored separately in a single file. During an experiment, the audio reading is triggered by the microphone continuously capturing audio at 16,000 Hz. The real-time captured audio data is then replaced with the pre-recorded audio. To synchronize between video and audio, the image feeding speed needs to be adapted: if the audio goes faster than video, some pre-recorded images need to be skipped; if the audio goes slower than video, the video feeding thread needs to wait. As an example, capturing at 15 FPS, if the camera is turned off for ten seconds and turned back on again, about 150 pre-recorded frames need to be skipped.

In Section 5.3, the experiments will require loading both accelerometer values and images. The time synchronization between those two sensor streams are done in a similar approach. The



C1: always transmit images for processing
 C2: image processing triggered by audio analysis done on cloudlet
 C3: image processing triggered by audio analysis done on mobile device

Figure 5.1: Example Sequence of Ping-pong Feedback over Time in Different Configurations

pre-recorded accelerometer values are stored in a single file, with timestamps attached to each value, and then used to replace real-time sensed acceleration data. The image sensing thread then adjusts the image loading rate based on how fast the acceleration data are read.

I have experimented with the effectiveness of sensor control on both the phone and Google Glass. For both devices, I was able to read real-time current and voltage values from system files¹² which are used to calculate power. I have verified that this measurement technique is accurate, consistent with the battery capacity specifications.

5.2.4 Evaluation

Table 5.2 compares the power consumption, average current, and associated battery life with three system configurations. The first configuration always captures and transmits image data, but the microphone remains off. In the second configuration, the system continuously transmits audio data to the cloudlet for analysis, and only turns on camera when ping-pong sound is detected. Finally, the third configuration is similar to the second, except that audio analysis is done natively on the mobile device.

Clearly, having application-specific sensor control significantly reduces the power consumption for Ping-pong. When the audio data analysis is offloaded to the cloudlet, the battery life

¹For Google Glass: `/sys/class/power_supply/bq27520-0/current_now`
 and `/sys/class/power_supply/bq27520-0/voltage_now`

²For Nexus 6 Phone: `/sys/class/power_supply/battery/current_now`
 and `/sys/class/power_supply/battery/voltage_now`

can be extended by 23% (on the Glass) to 30% (on the phone) when running the application. Having audio processing locally further conserves energy by 5% (on the Glass) to 16% (on the phone). This is because although additional load is put on the CPU, there is no need for audio data transmission to and from the cloudlet, saving power from WiFi usage.

The energy win comes at a small cost of delayed response during the beginning of a rally. Figure 5.1 shows the guidance a user gets (marked as vertical lines) in a ping-pong game for about one and a half minute. The different clusters indicate different rallies. The green and red marks indicate when the client receives signal to turn on or off the camera, respectively. As we can see, for each rally, the client almost always gets the command to turn on the camera as soon as the rally starts, indicating the responsiveness of the ping-pong sound detection technique. However, there may be a small delay between when the command is received to when the first guidance can be delivered. This is probably due to the fact that the camera needs some time to initialize and stabilize internal sensor circuitry. For example, some cameras need to go through the initial regulator stabilization period after canceling standby mode, and have to wait for 8 frames of initialization period before a normal image can be captured [3]. However, once the camera is initialized and the user starts to get guidance, the end-to-end latency of each guidance remains unchanged compared to the original implementation without sensor control. In addition, the delay will be reduced with better camera hardware that is faster in the initialization process.

5.3 Application-specific Sensor Control: Lego

5.3.1 Methodology

Unlike Ping-pong, Lego provides step-by-step instructions. After each guidance is given, a user needs some minimal amount of time (several seconds) to complete the step. During these periods of user actions, the camera can be turned off, and no Lego state analysis needs to be computed. Only when a user finishes a step do we need to turn on the camera, extract the symbolic Lego state, and provide next step instructions. But how can the application know when the user has finished one assembly step?

Policy 1

The first idea is similar to that used in Ping-pong, which uses the cheap sensor to continuously monitor a user's high-level state, and trigger more power-hungry sensors for further processing. In Lego, the accelerometer can be used as the cheap sensor. This is because the user's head is usually moving around (e.g. searching for Lego bricks) when doing assembly, leading to a higher acceleration variation. But once the user has finished a step and is waiting for the next instruction, their heads tend to stabilize, thus the accelerometer values are more constant. Therefore, the variation of acceleration values can be a good indicator to trigger turning on and off the camera for complex visual processing.

This leads to the first sensor control policy for Lego:

- Always sense and process the acceleration values to calculate its short term variation.
- Turn off camera as soon as a specific guidance is provided.
- Turn on camera as soon as the variation of acceleration values is below a threshold.

The acceleration variation calculation is done for every half a second. Since it is a very simple process, it can be either offloaded to the cloudlet, or done locally on the device, similar to the ping-pong sound detection.

Policy 2

Since the user needs at least several seconds to process a step, why do we bother processing any sensor data during that period at all? This is the intuition behind the second approach. It does not use a second sensor to predict when the image sensor needs to be activated. Instead, the camera will be turned off for a certain period of time once each guidance is provided. It will then be turned back on and continuously monitor the user's progress until the next guidance is provided, which sends it to doze state again.

Of course, how long a user takes to complete each step is not known a priori. We can only guess the duration and let the system rest based on the estimation. This estimation accuracy is critical. If the system rests too long, the user will not receive immediate guidance once she finishes a step. If the system rests too short, then it loses the opportunity to conserve more energy for the device. The next section describes how to make this estimation.

The sensor control policy for this strategy can be described as

- The camera is on initially.
- Turn off camera as soon as a specific guidance is provided.
- After a certain period of time (the duration to be defined in Section 5.3.2), the camera is turned back on.

In practice, the mobile device will receive two control commands that come with each guidance, one is to turn off the camera immediately, and the other is to turn on the camera with a specific delay, instructed by the `delay` field.

Policy 3

It is not hard to imagine that the previous two policies can be used in combination:

- The camera and accelerometer are on initially.

- Turn off both the camera and the accelerometer as soon as a specific guidance is provided.
- After a certain period of time (to be defined in Section 5.3.2), the accelerometer is turned back on.
- Turn on camera as soon as the variation of acceleration values is below a threshold.

I call this combination Policy 3, and Section 5.3.3 will compare the effectiveness of the three policies introduced above.

5.3.2 Estimating Step Time

Basic Approach

How can we accurately estimate each user's step time in Lego? The core idea is to use existing users' step time to predict the step time of a new user. In other words, if many users take longer time on a particular step, a new user will probably take longer time on the same step. This is because the time a user needs to complete each step largely depends on the difficulty of that particular action. For example, usually, the rarer the piece the user needs, the longer time it takes a user to find it. This difficulty level will affect all users in a similar way, thus there must be a strong correlation between the step time among different users. This intuition will be confirmed later in this section.

Based on this assumption, for each step, I will calculate the average time and standard deviation for all existing users. When a new user uses the application, the step time will be estimated as

$$t_{est_i} = \alpha \times t_{mean_i} + \beta \times t_{std_i} \quad (5.1)$$

where t_{est_i} is the estimated time for step i , t_{mean_i} is the average time of existing users for that step, and t_{std_i} is the standard deviation. α and β are parameters that could be tuned to trade between application responsiveness and energy saved. Since responsiveness is usually the top concern, they can be set to conservative values: α is below one, and β is negative.

Characterizing User Behavior

A better approach is to adapt the estimation based on user characteristics. For example, for a user that usually does assembly more slowly than others, the estimation should be larger, saving more energy of the system. The modified estimation is then

$$t_{est_i} = (\alpha \times t_{mean_i} + \beta \times t_{std_i}) \times user_scale \quad (5.2)$$

where $user_scale$ is used to quantify the relative speed of each user to other users. Before the first step, its initial value is set to 0.7 to be conservative. After the user completes each step i ,

	User 1	User 2	User 3	User 4	User 5
User 1	1	0.54	0.50	0.87	0.42
User 2	0.54	1	0.36	0.44	0.61
User 3	0.50	0.36	1	0.69	0.58
User 4	0.87	0.44	0.69	1	0.54
User 5	0.42	0.61	0.58	0.54	1

Interpreting the values:

Positive numbers: positive correlation; negative numbers: negative correlation.

Absolute values of 0-0.3: weak correlation; 0.3-0.7: moderate correlation; 0.7-1: strong correlation [10].

Table 5.3: Pearson Correlation [30] Coefficient Matrix of Step Time among Different Users

it is then updated based on the user’s step completion time and the average completion time of previous users, using the equation below

$$user_scale = user_scale \times p + \frac{t_step_i}{t_mean_i} \times (1 - p) \quad (5.3)$$

where t_step_i is the user completion time for step i . This is a standard moving average adaptation, and p is set to 0.7 in my implementation.

Evaluation of Step Time Estimation

To test the effectiveness of such estimation approach, I have asked five users to perform a couple of same Lego assembly tasks. The data collection process is the same as that used in Section 3.2.3. After they finish a Lego step, I will tell them what the next step is through verbal guidance, and show them the instructions using printed images. The whole process of them completing the task is videotaped by the Google Glass they wear on their heads. I then manually analyze the videos to get the actual step time of each user.

I first calculate the Pearson correlation [9, 30] of the step time among different users. This is a standard approach to evaluate the linear correlation between variables, with correlation coefficient values ranging from -1 to 1. A value above zero means positive correlation, and a larger number indicates stronger correlation. Table 5.3 shows the correlation coefficient matrix among all five users. All of the values are above 0.3, suggesting a moderate to strong correlation between any two users. This confirms the assumption behind Policy 2 and 3.

Next, I perform a cross-validation on these data. That is, each time, I select one user as the “new user” for testing, and use the other four users’ data for training. For each user, I measure three values, t_total , t_est_total , and t_delay . t_total is the total duration of the user performing the whole task.

$$t_total = \sum_{i=1}^N (t_step_i) \quad (5.4)$$

(a) Task 1

	t_{total} (s)	$t_{est_{total}}$ (s)	t_{delay} (s)	$t_{est_{total}}/t_{total}$ (%)
A1	79.8 (14.8)	35.3 (2.0)	0	45.4 (7.0)
A2		38.5 (7.1)	0	48.4 (4.2)

(b) Task 2

	t_{total} (s)	$t_{est_{total}}$ (s)	t_{delay} (s)	$t_{est_{total}}/t_{total}$ (%)
A1	101.2 (26.6)	38.8 (4.5)	0	41.0 (12.1)
A2		44.5 (11.1)	0	44.7 (5.5)

A1: without per-user adaptation (Equation 5.1)

A2: with per-user adaptation (Equation 5.2)

The numbers in parenthesis are standard deviations across users.

Table 5.4: Step Time Estimation for Lego

where N is the total number of steps for a task. $t_{est_{total}}$ is the summation of estimated time for each step. That is,

$$t_{est_{total}} = \sum_{i=1}^N (t_{est_i}) \quad (5.5)$$

In theory, this is the total time a system could “rest” (camera and networking turned off) in a task. If a system wants to be conservative to avoid any additional delay, it can then rest for only a portion of each estimated step time. Finally, t_{delay} is the extra delay, if any, this power conservation technique has incurred. This value is non-zero only when an estimated step time is larger than the actual step time. That is,

$$t_{delay} = \sum_{i=1}^N (\max(t_{est_i} - t_{step_i}, 0)) \quad (5.6)$$

Ideally, t_{delay} should be zero, while $t_{est_{total}}$ can be as close to t_{total} as possible.

Table 5.4 shows the measurement results for two tasks performed by the five users. To ensure $t_{delay} = 0$, I have set the parameters in Equation 5.1 to be $\alpha = 0.7$ and $\beta = -1.2$ and that in Equation 5.2 to be $\alpha = 0.8$ and $\beta = -1.2$, both tuned to be optimal. With this setting, the estimated step time is close to half of the actual step time, for both tasks. Allowing per-user adaptation only slightly improves estimation accuracy, which suggests the variation of assembly time among different users is small. If we turn off the camera and network for the period suggested by $t_{est_{total}}$, then it means the mobile device can be idle for close to half of the time during the application. Note that this does not mean we can save half of the energy, since the device consumes energy even in idle mode. The next section evaluates the real energy savings one can get through live experiments.

(a) Nexus 6 Phone

System Configuration	Power (W)	Current (mA)	Battery Life (h)
Always transmit video	2.9	843	3.82
Policy 1	1.6	444	7.26
Policy 2	2.1	540	5.97
Policy 3	1.6	398	8.08

(b) Google Glass

System Configuration	Power (W)	Current (mA)	Battery Life (h)
Always transmit video	2.5	702	0.81
Policy 1	2.1	567	1.01
Policy 2	1.7	446	1.28
Policy 3	1.5	390	1.46

Policy 1: uses acceleration variation to trigger image processing

Policy 2: system rests for each step based on step time estimation

Policy 3: system rests for each step based on step time estimation, and then uses acceleration variation to trigger image processing

Table 5.5: Power Consumption of Lego with Different Sensor Control Policies

5.3.3 Evaluation

All of the three policies have been implemented on top of the Gabriel platform, with camera and accelerometer control commands sent at appropriate times. I use all five users' data from Section 5.3.2 to train an estimator for step time estimation. A different video of Lego assembly is then used as the test video, along with the synchronized acceleration data. All other experimental setups are as described in Section 5.2.3.

Table 5.5 summarizes the energy consumption when different policies are used. When the images are continuously captured and streamed for processing, the application can run on the phone for 3.82 hours, but only 0.81 hours on Google Glass. When the acceleration variation is used as a signal for image processing (Policy 1), the power consumption is dropped dramatically from 2.9 W to 1.6 W on the phone, and from 2.5 W to 2.1 W on Google Glass, leading to 24.7% to 90.1% more battery life. Actually, with this setting, the camera is only woken up when the user has finished a step and waiting for guidance, so only a few frames need to be captured to analyze the user's state before the camera can go back to sleep again. Therefore, during the entire task of less than ten steps, only several tens of frames are captured and processed, leading to the huge energy win.

Using Policy 2 also significantly reduces power consumption (27.6% less on the phone and 32.0% less on the glass). As a combination of the previous two, Policy 3 performs the best. With this policy, the estimated step time allows the system to be idle for a significant portion of the total duration. For the rest time, accelerometer data are being consumed and processed, while only tens of image frames are captured and transmitted, similar to Policy 1. As a result, this

policy gives 8.08 hours battery life on the phone and 1.46 hours on the Google Glass, 111.5% and 80.2% better than the original configuration without sensor control, respectively.

5.4 Impact of Power Saving Mode on Latency and Energy

5.4.1 Background of Power Saving Mode

A lot of mobile devices support WiFi power saving mode (PSM) [13], which is a power-conservation technique originally defined in 802.11. With this technique, the mobile device will turn off its WiFi radio after a short period of network inactivity. The radio is then waken up if the device has anything to send, or to periodically check if the infrastructure (WiFi router) has queued any data item to deliver to it. The check period is usually hundreds of milliseconds and is a pre-set value determined by the access point and the device. Variants of this approach exist, such as the automatic power save delivery [1] and wireless multimedia extensions [14] specified in IEEE 802.11e. Besides WiFi, the cellular infrastructure usually employs similar techniques for power conservation, such as the discontinuous reception mode (DRX) [5, 33].

Such aggressive power management technique can incur additional network delay for applications, since the network receiver may not be awake as data arrive. In the wearable cognitive assistance application studied in this thesis, the mobile client sends sensor data to the cloudlet for processing, and then waits for its reply. If the server is able to quickly process the data and return the result before the device radio goes to sleep, the user will receive feedback messages immediately. However, if the server takes more than hundreds of milliseconds to process, the device radio will be turned down, and then be woken up only at slotted times to retrieve any data sent by the server. On average, this will incur half a slot time delay to the application response time.

Looking back to Figure 3.17, this extra delay is exactly the reason for the unexpectedly large blue components (feedback transmission time). For applications whose server processing is very fast (e.g. Pool and Ping-pong), the feedback transmission time is very low, since the network interface is always active. For some others (e.g. Lego and Drawing) whose server processing are much slower, the radio will enter low power state and may not pick up the server feedback immediately. Therefore, for these applications, it takes longer time to transmit server feedback, which is only a few tens of bytes, than to transmit images to the server, which is usually tens of kilobytes.

5.4.2 Tradeoff between Power and End-to-end Latency

One way to reduce the additional delay incurred by PSM is to have a background process running on the mobile device to keep the network radio awake, e.g. through continuous pinging of some server. This will, of course, eliminate the power savings provided by PSM. To quantify PSM's impact on latency and power for the wearable cognitive assistance applications, I have done experimentation with and without background pinging. To allow pinging intervals less than 200

(a) Lego

Configuration	Mean Power (W)	Mean Current (mA)	Battery Life (h)	Mean Latency (ms)
W/o pinging	2.81 (0.03)	737 (8)	4.37 (0.05)	315.7 (10.4)
W/ pinging (2 ms)	3.09 (0.05)	835 (11)	3.86 (0.05)	273.6 (5.3)
W/ pinging (20 ms)	2.98 (0.02)	773 (2)	4.16 (0.01)	267.0 (4.5)
Selective pinging (20 ms)	2.88 (0.02)	750 (2)	4.30 (0.01)	265.7 (7.3)

(b) Pool

Configuration	Mean Power (W)	Mean Current (mA)	Battery Life (h)	Mean Latency (ms)
W/o pinging	2.87 (0.02)	796 (14)	4.04 (0.07)	69.6 (0.7)
W/ pinging (2 ms)	3.11 (0.01)	891 (4)	3.62 (0.02)	71.2 (0.5)
W/ pinging (20 ms)	2.87 (0.01)	828 (2)	3.89 (0.01)	69.0 (0.3)
Selective pinging (20 ms)	2.84 (0.04)	828 (9)	3.89 (0.04)	69.4 (0.4)

The experiments are done with Nexus 6 Phone. The numbers in parenthesis indicate the standard deviation across three different runs. Selective pinging is an approach to start pinging only when the application expects return messages, as described in Section 5.4.2.

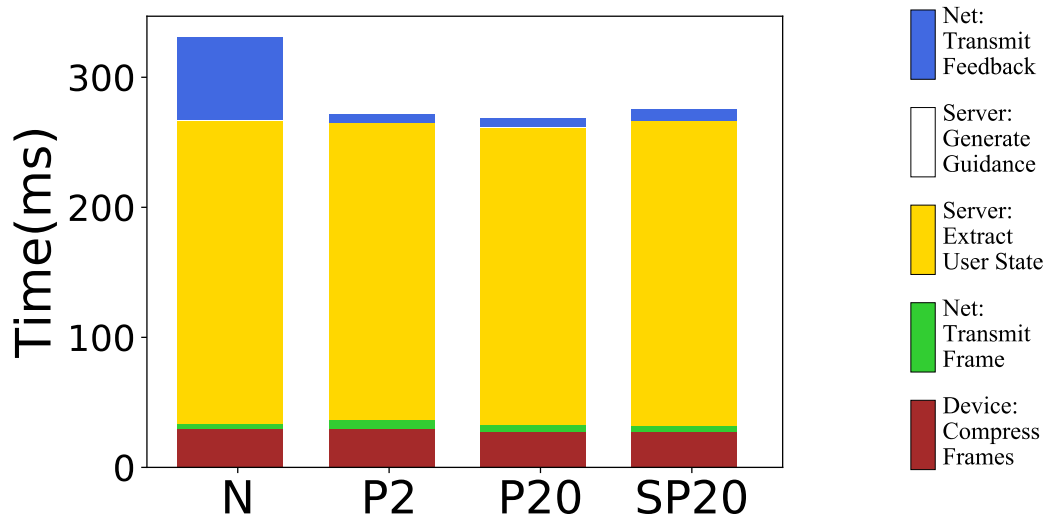
Table 5.6: Power Consumption and End-to-end Latency for Wearable Cognitive Assistance

ms, I had to use a rooted Android phone [11]. Similar to the experiments in Section 3.3.5, I used two applications as examples: Pool, which is the most interactive one, and Lego, which is one of the slowest.

Table 5.6 shows the end-to-end latency as well as power consumption, current, and associated battery life of the two applications with and without background pinging. Clearly, for Lego, fast pinging reduces the system response time by a nontrivial amount (15.4%). Figure 5.2 confirms that the improvement is from the reduction of feedback transmission time, since the only component that shrinks by disabling PSM is the blue part. This speed up comes at the cost of higher power consumption (10.0% more). On the other hand, disabling PSM has little impact on the latency and power consumption of Pool. This is because the fast server processing of Pool has already kept the radio of mobile device always on, and the background pinging process adds negligible overhead to the overall network traffic.

I have also experimented with different pinging intervals to explore the tradeoff between latency and power consumption. For Lego, increasing the interval from 2 ms to 20 ms will reduce power consumption by 3.6%, while still keeping the network transmission time low. This is because at 20 ms interval, the pinging traffic is sufficient to keep the radio awake, but incurs less traffic than pinging at 2 ms interval. In general, a higher pinging interval reduces power consumption, but will likely increase network delay. In my experience, pinging at 20 ms seems to achieve the best balance between latency and power consumption.

An even better approach of fine-grained pinging control is to proactively wake up the radio



N: without pinging
P2: pinging at 2 ms interval
P20: pinging at 20 ms interval
SP20: pinging at 20 ms interval only when return message is expected

Figure 5.2: Latency Breakdown of Different Background Pinging Mechanisms in Lego

only when the device is expecting a return message. For example, in Lego, the complete processing of an image is about 250 ms. Therefore, I let the device sleep for about 200 ms after sending each frame, and then start pinging at 20 ms interval to make sure the radio is on. The pinging thread terminates as soon as the return message is received, and will start again 200 ms after sending the next frame. This leads to the result in Table 5.6 named “selective pinging”. It further reduces the power consumption from 20 ms fixed interval pinging by 3.3%, while still achieving very low latency.

5.5 Chapter Summary

This chapter studies different energy conservation techniques in the context of wearable cognitive assistance. It shows that mobile sensors can be used more efficiently by using cheap sensors to trigger more expensive ones, or with accurate estimation of user action times. By experimenting with two example applications, I show that the battery life of mobile devices can be extended by 23% to 111% through careful sensor control. In addition, this chapter studies the tradeoff between power consumption and application response time introduced by system-wide optimization techniques, such as power saving mode.

Chapter 6

Related Work

This chapter describes the research projects and industrial effort related to this thesis. I start with listing existing mobile applications that serve similar purposes to wearable cognitive assistance in Section 6.1. These applications may be characterized using different terms, such as augmented reality, cognitive assistance, or virtual coaches, but they all try to deliver appropriate guidance to a user using a mobile device. Among these applications, many need to reach server infrastructure for accessing additional compute power or fetching appropriate data. The relevant work of such application offloading ideas will be discussed in Section 6.2.

One key requirement of many mobile applications is to ensure crisp system interactivity and responsiveness. But how fast is fast enough? As mentioned in Chapter 3, the answer largely depends on the application characteristics. Section 6.3 summarizes the results of latency sensitivity of a variety of mobile applications, some of which directly helped us to derive the target latencies in Chapter 3.

In Section 6.4, I will introduce recent research on speeding up computer vision algorithms and how they relate to the black-box multi-algorithm approach proposed in Chapter 4. Some of them are similar to my approach but have different focuses. Some others are orthogonal approaches that can be used in addition to the multi-algorithm technique for additional speed-up. In particular, I will mention many state-of-the-art solutions to speed up neural network based algorithms, due to its recent popularity in the computer vision community.

Finally, Section 6.5 discusses the efforts in conserving energy consumption on a mobile or wearable device, which has been a hot research topic for many years. The work in Chapter 5 draws insights from many of these efforts, and explores different techniques in the specific context of wearable cognitive assistance.

6.1 Wearable Cognitive Assistance

There have been many research efforts dedicated to offer cognitive assistance through a wearable device. For example, Siewiorek et al. [157] describe Virtual Coaches that provide proac-

tive assistance to handicapped people using sensor data (e.g. pressure sensor). Opportunity Knocks [139] offers navigation assistance to help those with mild cognitive disabilities. In the context of tourism, the GUIDE project [42] delivers relevant information to tourists based on their profile and location. More recent mobile augmented reality applications such as [165] can also recognize popular visual landmarks and scenes in the city that are stored in the system database and offer additional information. The Overlay system [97] makes one step further by allowing users to easily annotate new scenes and update the database.

In addition, a large portion of the work provides cognitive assistance for the visually impaired. For example, Chroma [167] uses Google Glass to provide color adjustments for the color blind. CrossNavi [153] uses smart phone to guide blind people to cross the road. VizLens [71] helps the visually disabled to easily interact with physical interfaces such as microwave control panels.

Furthermore, there exist many commercial applications based on smart phones or tablets that assist people's lives from different aspects. For example, the popular phone assistants such as Siri[26], Google Assistant [60], and Cortana [124] provide a speech interface to easily query relevant information such as weather, traffic, and agenda. Word Lens [176], which was later acquired by Google and integrated into Google Translate [62], can automatically translate the text being captured by a user's phone into a desired language. Recently, Google announced GoogleLens [66], which is similar to its earlier product, Google Goggles [63], that can provide near-real-time analysis of the image captured by a smartphone, such as recognizing QR codes, understanding texts, and detecting relevant objects.

Most of these applications mentioned above run completely on the mobile device (e.g. [62, 153, 167, 176]). As a result, their functionality are constrained by the limited computing power and battery capacity of the device. For those applications providing advance features using resources out-side of the wearable device (e.g. [26, 42, 60, 63, 66, 124, 153]), crisp response has not been a requirement. This thesis targets at enabling a new genre of wearable applications that need both high computation and crisp system response, which requires careful design of mobile-cloud system architecture and optimization of both network transmission and algorithm computation. In this regard, VizLens [71] and Overlay [97] are the most similar to this work, but they are using offloading solutions dedicated to their applications. In contrast, this thesis implements the general Gabriel platform that can support many different applications.

Finally, a different thread of work focuses on using wearable devices for memory rehabilitation. SenseCam [85] introduces a life logging system that uses a wearable camera to capture images in a user's daily life and later shown to the user for retrospective memory aid. People have also tried to use diverse sensor information such as audio [51] and location [101] to mediate memory. These efforts usually take a store-and-replay approach that requires no real-time analysis of the sensor information being captured. In contrast, the applications discussed in this thesis are much more interactive, requiring immediate assistance as the user is performing real world activities.

6.2 Cyber Foraging

Offloading compute-intensive operations from mobile devices to powerful infrastructure in order to improve performance and extend battery life is a strategy that dates back to the late 1990s [132, 149]. Flinn [56] gives a comprehensive survey of the large body of work in this space. The emergence of cloud computing has renewed interest in this approach, leading to commercial products such as Google Goggles [63], Amazon Silk [23], and Apple Siri [26]. Today, it is very convenient to start a new instance in the form of a VM, a container, or a compute engine in one of the popular cloud services, such as Amazon Cloud [24], Google Cloud [65], and Microsoft Azure [17]. Soon, similar services will also be available in the edge computing infrastructure. I refer to such edge instances as cloudlets in this thesis, but they are also known as micro data centers [69], fog nodes [32], or mobile edge cloud [35].

Managing the cloudlet infrastructure is more complex than traditional cloud computing, due to its distributed nature. In his dissertation [72], Ha has identified and implemented several core functionality that a cloudlet needs in addition to standard cloud computing components. The first component is to *discover a cloudlet*, which has the ideal network condition, computation load, and cache state to serve a particular user. The second component is *just-in-time VM provisioning* [74], which rapidly synthesizes the VM image of the back-end of an application and launches a VM instance of that image. Third, if a user moves while running an application, the back-end VM might need to migrate to different cloudlets using the *adaptive VM hand-off* [75] technique, which ensures network proximity between application front-end and back-end. These components provide infrastructure support to associate a user with the appropriate cloudlet. The Gabriel platform described in this thesis builds on top of this infrastructure, and directly supports different applications.

Given a specific offloading site, many projects have been exploring techniques and developing programming frameworks to efficiently partition an application and decide what and when to offload computation to the server for optimal speed, energy consumption. For example, Flinn et al. [58] introduce a self-tuning framework to monitor an application's resource need and available resources, and achieve a good balance between performance, energy usage, and quality of an application. Balan et al. [28] describe a programming language to lower the barrier of modifying existing applications to leverage remote resources. Among recent research efforts, MAUI [46] dynamically selects a subset of methods of a C# program based on the .NET framework to reduce the energy consumption of a mobile application. CloneCloud [44] dynamically migrates partial computation in the application VM (e.g. Java VM) between the mobile device and the cloud to improve application processing time and energy consumption. ThinkAir [107] builds on top of these two work, and further enables dynamic resource allocation and flexible parallel execution in the cloud environment. In addition, COMET [68] uses distributed shared memory to enable transparent offloading of Android applications. Odessa [142] makes dynamic offloading and parallelism decisions to achieve low latency for interactive perception applications. All of these research efforts constrain software to specific languages, frameworks or platforms. In contrast, the system architecture described in this thesis is capable of supporting the widest possible range of applications without restrictions on their programming languages or operating systems.

6.3 Latency Sensitivity of Mobile Applications

In mobile and ubiquitous computing, keeping system response time low is a key design consideration in building user-friendly applications. This dates back to studies of the latency tolerance of human users in the era of timesharing. Robert Miller's 1968 work on response time requirements [125] was an example. The recent advent of cloud computing and the associated use of thin-client strategies have made latency a bigger concern. In 2006, Tolia et al. [170] and Lai et al. [108] reported on the impact of latency on thin-client user experience. In 2015, Taylor et al. [169] contrasted the relative sensitivity of users to bandwidth and latency limitations for a cloud-sourced thin-client and thick-client computing. For applications targeting public displays, Clinch et al. [45] studied how the use of cloudlets could help support a spontaneous use of interactive applications. There have also been studies of the impact of latency in web-based systems, including those of Hoxmeier et al. [87] and Nah [128].

In the world of virtual reality (VR) applications, Ellis et al. [54] reported that VR applications that use head-tracked systems require latencies less than 16 ms to achieve perceptual stability. Watson et al. and Pavlovych et al. [140, 180] have also reported that users would achieve the best performance in a virtual environment when the latency is below a few tens of milliseconds. In addition, they have shown that variations of latency incur slight negative impact on human performance given a fixed mean latency.

To the best of my knowledge, little work has been reported to investigate the latency sensitivity of the emerging wearable cognitive assistance applications, where a user follows the guidance from wearable devices for task completion. This thesis takes a step to quantify the tolerable latency for several example applications. As mentioned in Section 3.2, the numbers provided are not strict bounds, but can be used as reference targets for designing new applications and systems.

6.4 Speeding up Computer Vision

6.4.1 Leveraging Hardware Support

Without dramatic modification to an algorithm, the most effective way to speed it up is through exploitation of its internal parallelism and to leverage special hardware such as ASICs, GPUs and FPGAs. For example, Huang et al. [89] has designed a hardware accelerator to speed up SIFT [119] feature extraction. More recently, the enormous computation involved in a deep neural network (DNN) has made dedicated accelerators like GPUs almost essential. NVIDIA and AMD are the biggest players in the GPU market, but many other companies have also been developing customized accelerators, including Intel (who has acquired Movidius that creates VPUs [96]), Google (TPUs [61]), Xilinx [181] and Deephi [49] (FPGAs), and Huawei (Kirin NPUs [90]). However, not all algorithms can benefit from the use of such specialized hardware, since the improvement is limited by the degree of internal parallelism. In addition, such approaches usually require a rewrite of the relevant parts of existing algorithm code. Several techniques have been proposed to simplify the process to leverage parallelism. For example,

Halide [143] provides a cross-platform language and compiler for easy experimentation of image processing workflows. However, these work usually impose restrictions on how an algorithm is structured.

6.4.2 Multi-algorithm Approaches

Another thread of work has been trying to leverage a coarser granularity of parallelism. In these approaches, multiple algorithms are run concurrently, and selectively use the result from one of them to strike a balance between speed and accuracy. For example, since the early impactful face detection system by Viola and Jones [175], a substantial body of work has been using a cascade approach to speed up image classification, object detection, and OCR tasks [40, 70, 161, 173]. In these approaches, an input image goes through multiple algorithms one by one. If, for any algorithm, it is recognized with high confidence, the result will be used and it will not be fed into later algorithms, saving the time and computation resources. The thresholds regarding confidence scores can be set for each algorithm to control its pass rate, thus achieving different speed-accuracy tradeoff. Note that although in their original form, all the algorithms are run sequentially, it is possible to run them concurrently given sufficient computing power, thus further speeding up the process.

Although the cascade approach has been helpful, it fails to leverage temporal locality of a video stream. Moreover, some of these approaches require every algorithm to have a confidence score to filter out samples, which may not be the case for some algorithms. In contrast, the multi-algorithm approach proposed in this thesis takes advantage of the temporal coherence between nearby frames, and treats each algorithm as a pure blackbox.

6.4.3 Optimization for DNNs

Because of the popularity of DNN based computer vision techniques, there are quite a few recent efforts dedicated to speeding up such algorithms. For example, model compression [27, 78, 92] is a technique that compresses an existing neural network model to a usually shallower architecture with less parameters to achieve faster speed with minimal sacrifice in accuracy. Model quantization [91] constrains the size of each parameter to be a few bits to reduce total model size for faster processing and less power consumption. Model distillation [36, 83] targets compressing the knowledge of an ensemble of expert models into a single model with the same accuracy level and acceptable model complexity. Recent work on network slimming [117] automatically identifies insignificant connections in a large neural network and prunes it to a more compact one. Finally, MCDNN [77] explores the use of above model compression techniques in a mobile and cloud computing environment with limited compute, memory, and energy budget. All of these approaches are orthogonal to the multi-algorithm approach proposed in Chapter 4: one can use one of the above approaches to speed up a particular DNN model, and then combine multiple different algorithms to achieve an additional speed-up.

In addition, Deep Feature Flow [187] specifically targets speeding up DNN processing on video streams. Similar to my multi-algorithm approach, it leverages the temporal locality of

video content, and tries to reuse partial processing result from the previous frame to process the current frame. However, this approach depends deeply on the particular model characteristics of a neural network, and cannot be extended to other computer vision approaches. In contrast, the multi-algorithm approach proposed in this thesis is a black-box meta-algorithm that can be applied to any combination of algorithms.

6.5 Energy Conservation on Mobile Devices

A lot of work has been done to characterize the energy consumption of mobile devices, and these have shown that processing (e.g. CPU), network transmission (e.g. WiFi), sensor reading (e.g. GPS, camera), and display are the biggest energy consumers in typical mobile applications [115, 154]. Therefore, a variety of approaches have been proposed to optimize energy usage by these components. For example, LiKamWa et al. [114] has proposed techniques to make the energy consumption of image sensors proportional to their sensing rate and quality. This technique can be used on top of the approaches proposed in Chapter 5 to further reduce energy consumption. In addition, RedEye [116] applies customized circuitry in the early phase of image sensing pipeline to allow continuous processing of neural networks on the mobile device. StarFish [113] optimizes for efficient sharing of computation among concurrent vision applications on a smart glass. Kobe [43] is a tool to simplify the development of mobile classifiers and achieve an optimal energy-accuracy-speed tradeoff. All of these work assume complex processing on the mobile device, thus cannot be applied directly to the cloudlet-offload processing model in the Gabriel platform. However, they can be used in combination with Offload Shaping [88], where a small amount of processing is needed on the mobile device to wisely select the sensor data to be offloaded for energy and bandwidth reduction.

Many other efforts have focused on reducing power consumption for a particular type of mobile applications, including Email [182], Web browsing [38], and WiFi tethering [76]. Wang et al. [179] described a framework to simplify controlling the states and duty cycles of power-hungry sensors for user state recognition. These efforts inspired the work in Chapter 5, which focuses on reducing energy consumption on the emerging wearable cognitive assistance applications.

Chapter 7

Conclusion and Future Work

This thesis aims to enable a new genre of wearable cognitive assistance applications that are both computation intensive and latency sensitive. It proposes Gabriel, an application platform that simplifies the creation of such applications by factoring out common functionality. Through careful architectural design, the platform ensures low latency in network transmission, allows coarse-grain parallelism to speed up server computation, and provides simple interface for energy conservation on mobile devices. The ease-of-use and effectiveness of this platform is evaluated using nine prototype applications that I have implemented.

In this chapter, I will first detail the contributions made in this dissertation (Section 7.1). I will then discuss valuable directions for future work (Section 7.2). As stated at the beginning of the dissertation, recent technology advances have made today an exciting moment to start exploration of this new application space. I truly hope the remaining challenges can be solved, and people can start benefiting from such applications in the near future.

7.1 Contributions

This thesis claims that

A new genre of latency sensitive wearable cognitive assistance applications based on one or more mobile sensors can be enabled by factoring out common functionality onto an application platform. An essential capability of this platform is to use cloudlets for computation offloading to achieve low latency. The platform simplifies the exploitation of coarse grain parallelism on cloudlets, the conservation of energy on mobile devices, and re-targeting applications for diverse mobile devices.

In this dissertation, I have validated this thesis statement from multiple aspects. This section summarizes the major contributions of this work.

7.1.1 Identification of New Application Category

This thesis studies a new genre of *wearable cognitive assistance* applications that can help people in many aspects of life, including social interactions, cooking, medical training, industrial trouble shooting, furniture assembly, and so on. Although the general concept of using mobile devices for user assistance is not new, the applications studied in this thesis have a unique combination of requirements of both computation intensity and latency sensitivity, which makes them impossible to deploy with today's compute and network infrastructure. This dissertation identifies the unique challenges in enabling these applications, and argues that only with recent technology advances has this goal become attainable.

One way to think about this new class of applications is that it combines the delay intolerance of AR applications with the very high computational demands of AI applications. To distinguish them from previous AR applications, this dissertation creates a taxonomy of the AR eco-system based on the degree of immersion, computation complexity, and network delay tolerance. As far as I'm concerned, this is the first attempt to create a taxonomy of this application space.

7.1.2 Gabriel Platform for Wearable Cognitive Assistance

To allow fast development and experimentation of wearable cognitive assistance applications, I develop the Gabriel platform, which factors out many of the functionality shared among applications, including networking, data encoding, and component discovery. With this platform, the creation of new applications requires only the implementation of application-specific logic. Gabriel is an open-source project at <https://github.com/cmusatyalab/gabriel>. So far, its client front-end supports various Android devices (e.g. Nexus 6 Phone, Google Glass), HoloLens (running Windows), and a python emulator.

Core features of the Gabriel platform are highlighted below.

- Low end-to-end latency is one of the most critical requirements in the design of Gabriel. In order to avoid any implicit queuing in the OS, application, or networking stack, Gabriel has a built-in application-layer flow control mechanism.
- Gabriel offloads the heavy computation of computer vision and signal processing to a cloudlet, which is only one wireless hop away from the mobile device. The network proximity allows a mobile application to access server-class compute resources with very low latency and high bandwidth.
- A pub-sub backbone in the Gabriel back-end allows flexible data flow among different cognitive modules, which forms the basis for coarse-grain parallelism. Since each cognitive module is encapsulated in a virtual machine, it simplifies the reuse of existing software.
- Gabriel provides a simple interface for each application to control the mobile sensors, while the Gabriel front-end abstracts out the details in dealing with different devices and operating systems. This provides an opportunity for energy conservation on the device.

7.1.3 Prototype Application Implementation

I implement nine prototype applications for wearable cognitive assistance on top of the Gabriel platform. These applications cover a broad range of application scenarios. Some of them provide step-by-step instructions, while some others are more interactive, providing responses to specific events. They also use a variety of computer vision and signal processing techniques, ranging from simple color detection, to the state-of-the-art neural-network-based object detection and face recognition. Because of the diversity of these applications, they serve as a good benchmark suite for system analysis.

On top of the implementation, I also quantify the target latencies for each of these applications using different approaches, including literature search, first principles of physics analysis, and user study. These target latency numbers provide a way to translate system latency measurement numbers to user experience. They offer clear guidelines in building such applications and systems.

7.1.4 Empirical Study of End-to-end Latency

I carry out an empirical study to analyze the latency performance of the prototype applications under different system configurations, including different offloading sites, mobile hardware, and wireless networks. These analysis and experiments provide constructive insights for designing new wearable cognitive assistance applications and the distributed infrastructure needed to support them.

In my study, I show that using a WiFi-connected cloudlet gives the ideal latency performance, because of the low network RTT and high bandwidth it offers. By comparing the measurement results with their target latencies, it is clear that cloudlets can help meet user expectation for most applications. Furthermore, 4G LTE-connected cloudlets can also offer fast responses significantly better than using public cloud. More surprisingly, while most of the applications offload back-end processing to the cloudlet or cloud, the choice of client device also greatly impacts end-to-end latency, due to differences in networking technology and encoding speed.

Furthermore, I log timestamps at critical points of an application, and break down end-to-end latency. Results show that with the help of cloudlets, both uplink and downlink networking time are dramatically improved, leaving the server computation time as the performance bottleneck. This should be the focus of further performance optimization.

7.1.5 Multi-algorithm Approach to Speed up Server Computation

While using additional CPU cores and specialized hardware accelerators can speed up the server computation of some applications, the benefit it provides depends largely on the specific algorithm and implementation being used. To further speed up more applications, I propose a black-box multi-algorithm approach. The core insights behind this approach are two folds. First, for any computer vision task, there usually exist many different algorithms, with different speed-

accuracy tradeoffs. Second, even for the less accurate algorithms, there exists temporal locality of accuracy when they are used to process a video stream.

My multi-algorithm approach exploits such temporal locality of accuracy. It runs multiple algorithms in parallel, and uses the more accurate (but slower) ones to evaluate the real-time accuracy of the faster (but less accurate) ones. It then adaptively selects which ones to trust and use based on their runtime performance. Using this approach, I have demonstrated significant speed-ups for three prototype applications with little impact on accuracy.

7.1.6 Conserving Power Consumption with Sensor Control

Big opportunities reside in reducing power consumption of the wearable cognitive assistance applications by carefully control the power-hungry sensors on the mobile device. The Gabriel platform provides a simple interface that allows each application to control the sensors with its own policy. For example, an application can send a command such as “turn off the camera for five seconds” to the platform, and the rest will be automatically taken care of.

I demonstrate the effectiveness of sensor control using two example applications, Lego and Ping-pong. Ping-pong uses a cheap sensor (microphone) to activate the more expensive one (camera) only when necessary. Lego employs a similar approach, while further rests the device based on assembly-step-time estimation. Results show that a carefully designed sensor-control policy can extend the battery life by 23% to 30% for Ping-pong, and by 80% to 111% for Lego.

7.2 Future Work

This thesis demonstrates the feasibility of wearable cognitive assistance with the help of cloudlets. To deploy these applications in the real world, there remain several challenges to address. This section discusses three directions that I find valuable as future work: improving system scalability, prototyping new applications, and better design of wearable hardware and software.

7.2.1 Scaling Wearable Cognitive Assistance

Cloudlet resource management

This thesis focuses on optimizing performance of wearable cognitive assistance applications for one user. As a result, the cloudlet resource was not a concern and was treated as unlimited. The Gabriel platform may spawn many VMs to run different cognitive modules for one application. Using the multi-algorithm approach introduced in Chapter 4, there could be even more VMs dedicated to the same task to reduce application response time. This is too heavy footprint for a real-world deployed cloudlet that serves multiple users concurrently. Therefore, careful management is needed for resources of compute, memory, and storage on the cloudlet.

In the Gabriel back-end, the Control VM is the single contact point on the cloudlet side for

every application instance, and the User Guidance VM runs application logic and may contain user specific states. So these two components should probably be kept as is, and instantiated for every user and application. However, the Cognitive VMs usually run general computer vision tasks, such as face detection and object detection, that are needed by many applications and users. Therefore, it is possible to treat these cognitive modules as micro-services [7] and share them among different users to reduce their system footprint.

One way to share the service is to let multiple users connect to the same server in a VM. However, this would require every service to implement its own performance isolation technique to ensure fairness among users. In addition, if any malicious user successfully breaks down the service, all other users will be affected. Therefore, a better approach is to let each service serve one user, but encapsulate them in lighter weight containers such as Docker containers for smaller footprint, performance isolation, and easy management of user state, if any. All these containers can then be grouped to run on top of VMs for better security isolation.

As a result, there needs to be a meta service running on each cloudlet to manage all these resources. Specifically, it needs to

- Monitor the usage of system resources, including CPUs, memory, disk, etc.
- Upon request, instantiate a new cognitive service in a container and try to find an existing VM that hosts the same containers to host it. If all such VMs are fully occupied, instantiate a new VM.
- Clean up the containers and VMs upon request or by periodically checking for zombie services.

Creating applications at scale

To attract more users to try out wearable cognitive assistance, more high-quality applications need to be developed. The prototype applications developed in this thesis are a good proof-of-concept, but the real deployable applications need to be more robust against corner cases and adapt well to different user behaviors. The Gabriel platform also needs to be better packaged into an easy-to-use SDK to simplify the development of new features on both the server and the client side.

Although Gabriel has factored out many common functionality among applications, it doesn't help the creation of application-specific computer vision algorithms. In fact, developing these algorithms requires computer vision expertise, and can be very time consuming. The expertise is not only required to solve a specific task, but also to decompose the entire application into solvable computer vision tasks. For example, in the development of Lego, I knew at the beginning that some algorithms can be helpful, including line detection, color detection, etc. All these knowledge were learned from computer vision courses at CMU. However, even with these knowledge, it still took me more than four months to develop and polish Lego to make it robust, where the majority of time was spent on parameter tuning and testing in different lighting conditions.

Lowering the barrier of entry to development is the key to create these applications at scale. In

particular, some high-level computer vision programming frameworks and tools will be helpful for developers with less experience in this domain. An example of such tools is an easy-to-use GUI to label user-specific images, and then to generate object detectors without writing computer vision code. In addition, it would be great if an intelligent system can automatically detect task-relevant objects, actions, and patterns from demonstration videos of experts completing the task. This, in combination with automatic generators of object, action, and pattern detectors, may automate the creation of complete application back-ends. This approach actually follows the learning habit of human beings: we can watch a tutorial video (even without sound) and easily copy the procedure for many tasks.

Real-world deployment and test

The experiments done in this thesis are in a lab environment with only several users using the cloudlet at the same time. To truly understand the application performance at scale, and to evaluate the resource management techniques mentioned above, a real-world cloudlet testbed is needed. The Living Edge Lab [95] built by the Open Edge Computing initiative [18] is a good first step. It allows users with phones of certain frequency bands to access cloudlet resources at Pittsburgh through 4G wireless hop. In the future, a more open environment is needed to allow more phones with a different frequency bands to participate. More testbeds are also needed to study the performance difference caused by various network interference, server hardware heterogeneity, and user habits of different populations.

7.2.2 Prototyping New Applications

Multi-sensor applications

Although the Gabriel platform supports processing multiple sensor streams concurrently, the prototype applications introduced in this dissertation all focus on visual processing only. Chapter 5 does use different sensors, but they were used purely for energy saving purpose. It will be nice to further diversify the benchmark suite with applications that leverage multiple sensors. For example, in a industrial trouble-shooting use case, real-time analysis can be made on both video streams as well as the sound from tapping an equipment. Furthermore, if the equipment has embedded acceleration or pressure sensors, their sensor streams can be analyzed by the Gabriel back-end as well, since Gabriel is able to handle multiple devices concurrently, as mentioned in Section 2.2.5.

Multi-user use cases

The application scenarios discussed so far are single-user applications. More complicated cognitive assistance use cases may require cooperation among multiple users in real-time. Figure 7.1 presents some hypothetical examples that are natural extensions to existing applications. To support these new use cases, the Gabriel platform needs to be modified to share real-time states

among different users. In the multi-user Furniture example, this includes synchronizing the assembly steps between the two users, and ensuring that they get appropriate guidance for efficient collaboration.

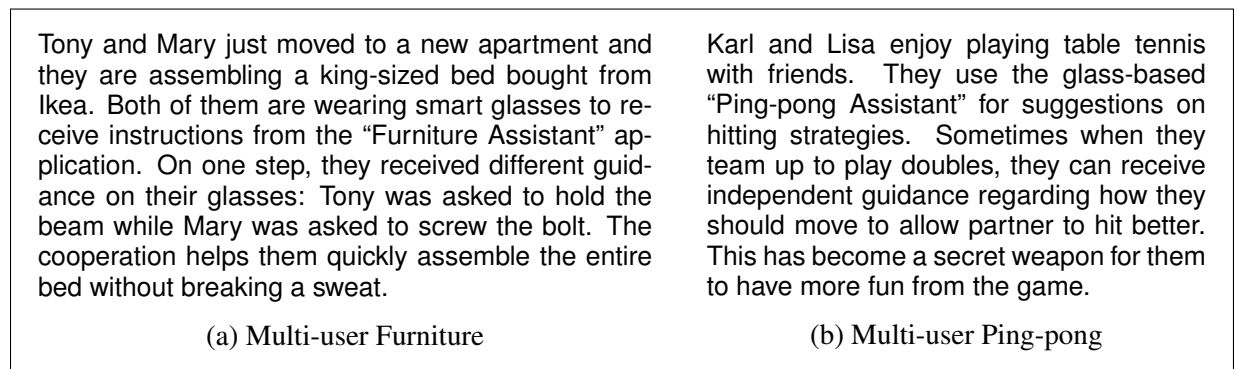


Figure 7.1: Wearable Cognitive Assistance Use Cases for Future Work

Two approaches to extend the Gabriel platform are possible for such state sharing. The first is to allow one Control VM and one User Guidance VM to serve multiple users so that all application states are contained within the VMs. This will complicate the implementation of the two VMs, which have to deal with streams from multiple users. In addition, since the Control VM will be launched once the first user starts the application, careful attention needs to be given once the rest users join the app to ensure they connect to the correct Control VM. The second approach is to keep the current architecture intact, in which every user has her own Control VM. An extra communication mechanism then needs to be developed to allow different Control VMs to discover each other, and to efficiently share states among them.

7.2.3 Better Mobile Hardware/Software Support

Although wearable hardware is improving at a dramatic pace, their comfortness and functionality are not yet ready for a wide market adoption. In addition, little efforts have been made to develop system software to support the new wearable cognitive assistance applications. Below I identify three areas that need to be improved on mobile devices.

- None of the existing devices have achieved a satisfactory balance between display quality and comfort. For devices that are relatively small and lightweight (e.g. Google Glass), the display is usually too small to display text or high resolution image. For devices that have better screen quality such as HoloLens, the bulky size and weight prevent people from wearing them for a long time. In addition, these devices usually do not work well with existing prescription glasses. Therefore, significant improvement in hardware design is needed to make such devices comfortable to wear for everyday tasks.
- In this thesis, one aspect of wearable cognitive assistance that is not studied is the human-computer interaction (HCI) interface design. What is the best way to provide guidance for such applications, video, image, or auditory text? If anything is unclear to the user, what

is the best way to take users' inputs, gestures or voice? These are all questions that need to be answered for a better user experience.

- Despite the efforts made in Chapter 5, the energy performance is still not good enough for some applications and devices. Even if an application can last long enough, the accumulated heat on device may cause user discomfort and eventually slow down the system processing speed. For example, many users who have tried Ping-pong have reported that the Google Glass becomes too hot after several minutes of usage. Clearly, more efforts are needed to reduce power consumption and to improve thermal dissipation of the devices.

Bibliography

- [1] Automatic power save delivery. https://en.wikipedia.org/wiki/IEEE_802.11e-2005#Automatic_power_save_delivery. Accessed on October 25, 2017. 97
- [2] Augmented reality pool. <http://rcvlab.ece.queensu.ca/~gridb/ARPOOL.html>. Accessed on November 27, 2015. 36
- [3] Sony imx225lqr datasheet. http://astroccd.org/wp-content/uploads/2016/01/IMX225LQR-C_E_TechnicalDatasheet_Rev0.2.pdf. Accessed on November 3, 2017. 91
- [4] Cloudlet launcher. <https://play.google.com/apps/testing/edu.cmu.cs.elijah.cloudletlauncher>. Accessed on March 28, 2018. 24
- [5] Discontinuous reception mode. https://en.wikipedia.org/wiki/Discontinuous_reception. Accessed on November 2, 2017. 97
- [6] Fractional ball aiming. <http://billiards.colostate.edu/threads/aiming.html#fractional>. Accessed on November 27, 2015. 36
- [7] Micro services. <https://en.wikipedia.org/wiki/Microservices>. Accessed on November 9, 2017. 111
- [8] Opencv library. <http://opencv.org>. Accessed on July 21, 2017. 30, 36
- [9] Pearson correlation coefficient. https://en.wikipedia.org/wiki/Pearson_correlation_coefficient, . Accessed on November 5, 2017. 94
- [10] How to interpret a correlation coefficient? <http://www.dummies.com/education/math/statistics/how-to-interpret-a-correlation-coefficient-r/>, . Accessed on November 6, 2017. 94
- [11] Rooting (android). [https://en.wikipedia.org/wiki/Rooting_\(Android\)](https://en.wikipedia.org/wiki/Rooting_(Android)). Accessed on November 5, 2017. 98
- [12] Cloud tpus - ml accelerators for tensorflow. <https://cloud.google.com/tpu/>. Accessed on August 19, 2017. 4
- [13] Wifi power saving mode. https://wiki.maemo.org/Wifi_power_saving_mode. Accessed on October 25, 2017. 97
- [14] Wireless multimedia extensions. <https://en.wikipedia.org/wiki/>

Wireless_Multimedia_Extensions. Accessed on October 25, 2017. 97

- [15] Issue 512: Bluetooth connection interfering with TCP Traffic on WiFi. <https://code.google.com/p/google-glass-api/issues/detail?id=512>, May 2014. 53
- [16] Base 64 Encoding and Decoding. <https://en.wikipedia.org/wiki/Base64>, 2017. 23
- [17] Microsoft azure: Cloud computing platform and services. <https://azure.microsoft.com/en-us/>, 2017. 4, 103
- [18] Open edge computing. <http://openedgecomputing.org/>, 2017. 4, 112
- [19] 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. 4
- [20] Sharad Agarwal, Matthai Philipose, and Paramvir Bahl. Vision: the case for cellular small cells for cloudlets. In *Proceedings of the fifth international workshop on Mobile cloud computing & services*, pages 1–5. ACM, 2014. 62
- [21] Trevor R Agus, Clara Suied, Simon J Thorpe, and Daniel Pressnitzer. Characteristics of human voice processing. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 509–512. IEEE, 2010. 10
- [22] Timo Ahonen, Abdenour Hadid, and Matti Pietikainen. Face description with local binary patterns: Application to face recognition. *IEEE transactions on pattern analysis and machine intelligence*, 28(12):2037–2041, 2006. 73
- [23] Amazon. Amazon Silk. https://en.wikipedia.org/wiki/Amazon_Silk, 2011. 103
- [24] Amazon. Amazon web services (aws) - cloud computing services. <https://aws.amazon.com/>, 2017. 4, 103
- [25] Asm Iftekhar Anam, Shahinur Alam, and Mohammed Yeasin. Expression: A dyadic conversation aid using google glass for people with visual impairments. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct Publication*, pages 211–214. ACM, 2014. 39
- [26] Apple. Siri. <https://www.apple.com/ios/siri/>. Accessed on October 5, 2017. 102, 103
- [27] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in neural information processing systems*, pages 2654–2662, 2014. 105
- [28] Rajesh Krishna Balan, Darren Gergle, Mahadev Satyanarayanan, and James Herbsleb. Simplifying cyber foraging for mobile devices. In *Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 272–285. ACM, 2007. 103
- [29] Peter N. Belhumeur, João P Hespanha, and David J. Kriegman. Eigenfaces vs. fisherfaces: Recognition using class specific linear projection. *IEEE Transactions on pattern analysis*

and machine intelligence, 19(7):711–720, 1997. 67, 68, 73

- [30] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009. xvii, 94
- [31] Adam L Berger, Vincent J Della Pietra, and Stephen A Della Pietra. A maximum entropy approach to natural language processing. *Computational linguistics*, 22(1):39–71, 1996. 13
- [32] 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. 15, 103
- [33] Chandra S Bontu and Ed Illidge. Drx mechanism for power saving in lte. *IEEE Communications Magazine*, 47(6), 2009. 97
- [34] Reinoud J Bootsma and Piet C Van Wieringen. Timing an attacking forehand drive in table tennis. *Journal of experimental psychology: Human perception and performance*, 16(1):21, 1990. 47, 48
- [35] Gabriel Brown. Converging Telecom & IT in the LTE RAN. White Paper, Heavy Reading, February 2013. 3, 15, 58, 103
- [36] Cristian Bucilu, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–541. ACM, 2006. 105
- [37] Jose M. Buenaposada and Baumela Luis. Variations of Grey World for face tracking. In *Image Processing & Communications 7*, 2001. 40
- [38] Duc Hoang Bui, Yunxin Liu, Hyosu Kim, Insik Shin, and Feng Zhao. Rethinking energy-performance trade-off in mobile web page loading. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 14–26. ACM, 2015. 106
- [39] Ken Chatfield, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Return of the devil in the details: Delving deep into convolutional nets. *arXiv preprint arXiv:1405.3531*, 2014. 79
- [40] Kumar Chellapilla, Michael Shilman, and Patrice Simard. Combining multiple classifiers for faster optical character recognition. In *International Workshop on Document Analysis Systems*, pages 358–367. Springer, 2006. 105
- [41] Ming-yu Chen and Alexander Hauptmann. *Mosift: Recognizing human actions in surveillance videos*. PhD thesis, Carnegie Mellon University Pittsburgh, PA, 2009. 30
- [42] Keith Cheverst, Nigel Davies, Keith Mitchell, and Adrian Friday. Experiences of developing and deploying a context-aware tourist guide: the guide project. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 20–31. ACM, 2000. 102
- [43] David Chu, Nicholas D Lane, Ted Tsung-Te Lai, Cong Pang, Xiangying Meng, Qing Guo, Fan Li, and Feng Zhao. Balancing energy, latency and accuracy for mobile sensor data classification. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor*

Systems, pages 54–67. ACM, 2011. 106

- [44] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011. 103
- [45] Sarah Clinch, Jan Harkes, Adrian Friday, Nigel Davies, and Mahadev Satyanarayanan. How close is close enough? understanding the role of cloudlets in supporting display appropriation by mobile users. In *Pervasive Computing and Communications (PerCom), 2012 IEEE International Conference on*, pages 122–127. IEEE, 2012. 104
- [46] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010. 103
- [47] Martin Danelljan, Fahad Shahbaz Khan, Michael Felsberg, and Joost Van de Weijer. Adaptive color attributes for real-time visual tracking. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1090–1097, 2014. 82
- [48] Nigel Davies, Keith Mitchell, Keith Cheverst, and Gordon Blair. Developing a context sensitive tourist guide. In *1st Workshop on Human Computer Interaction with Mobile Devices, GIST Technical Report G98-1*, 1998. 1
- [49] Deephi. Deephi. <https://en.wikipedia.org/wiki/Xilinx>. Accessed on October 6, 2017. 104
- [50] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009. 46
- [51] Lina Dib, Daniela Petrelli, and Steve Whittaker. Sonic souvenirs: exploring the paradoxes of recorded sound for family remembering. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, pages 391–400. ACM, 2010. 102
- [52] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: The International Journal for Geographic Information and Geovisualization*, 10(2):112–122, 1973. 37
- [53] Richard O Duda and Peter E Hart. Use of the hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972. 36
- [54] Stephen R Ellis, Katerina Mania, Bernard D Adelstein, and Michael I Hill. Generalizability of latency detection in a variety of virtual environments. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 48, pages 2632–2636. SAGE Publications Sage CA: Los Angeles, CA, 2004. 104
- [55] Gerhard P Fettweis. A 5g wireless communications vision. *Microwave Journal*, 55(12): 24–36, 2012. 6
- [56] Jason Flinn. Cyber foraging: Bridging mobile and cloud computing. *Synthesis Lectures on Mobile and Pervasive Computing*, 7(2):1–103, 2012. xvii, 10, 11, 103

- [57] Jason Flinn and Mahadev Satyanarayanan. *Energy-aware adaptation for mobile applications*, volume 33. ACM, 1999. 12
- [58] Jason Flinn, SoYoung Park, and Mahadev Satyanarayanan. Balancing performance, energy, and quality in pervasive computing. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 217–226. IEEE, 2002. 103
- [59] Todor Ganchev, Nikos Fakotakis, and George Kokkinakis. Comparative evaluation of various mfcc implementations on the speaker verification task. In *Proceedings of the SPECOM*, volume 1, pages 191–194, 2005. 88
- [60] Google. Google assistant. <https://assistant.google.com/>, . Accessed on October 5, 2017. 102
- [61] Google. Tpu. <https://cloud.google.com/tpu/>, . Accessed on October 6, 2017. 104
- [62] Google. Google Translate. https://en.wikipedia.org/wiki/Google_Translate, . 102
- [63] Google. Google Goggles. https://en.wikipedia.org/wiki/Google_Goggles, 2011. 102, 103
- [64] Google. Google Glass. https://en.wikipedia.org/wiki/Google_Glass, 2012. 18
- [65] Google. Google cloud computing, hosting services & apis. <https://cloud.google.com/>, 2017. 4, 103
- [66] Google. Google Lens. https://en.wikipedia.org/wiki/Google_Lens, 2017. 102
- [67] Google. tesseract-ocr. <https://github.com/tesseract-ocr/tesseract>, 2017. 30
- [68] Mark S Gordon, D Anoushe Jamshidi, Scott Mahlke, Z Morley Mao, and Xu Chen. Comet: code offload by migrating execution transparently. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 93–106, 2012. 103
- [69] Kira Greene. AOL Flips on ‘Game Changer’ Micro Data Center. <http://blog.aol.com/2012/07/11/aol-flips-on-game-changer-micro-data-center/>, July 2012. 15, 103
- [70] Alexander Grubb and Drew Bagnell. Speedboost: Anytime prediction with uniform near-optimality. In *AISTATS*, volume 15, pages 458–466, 2012. 105
- [71] Anhong Guo, Xiang’Anthony’ Chen, Haoran Qi, Samuel White, Suman Ghosh, Chieko Asakawa, and Jeffrey P Bigham. Vizlens: A robust and interactive screen reader for interfaces in the real world. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*, pages 651–664. ACM, 2016. 102
- [72] Kiryong Ha. *System Infrastructure for Mobile-Cloud Convergence*. PhD thesis, Carnegie

Mellon University Pittsburgh, PA, 2016. 103

- [73] Kiryong Ha, Padmanabhan Pillai, Grace Lewis, Soumya Simanta, Sarah Clinch, Nigel Davies, and Mahadev Satyanarayanan. The impact of mobile multimedia applications on data center consolidation. In *Cloud Engineering (IC2E), 2013 IEEE International Conference on*, pages 166–176. IEEE, 2013. 15
- [74] 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. 32, 103
- [75] Kiryong Ha, Yoshihisa Abe, Thomas Eiszler, Zhuo Chen, Wenlu Hu, Brandon Amos, Rohit Upadhyaya, Padmanabhan Pillai, and Mahadev Satyanarayanan. You can teach elephants to dance: agile vm handoff for edge computing. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, page 12. ACM, 2017. 23, 103
- [76] Hao Han, Yunxin Liu, Guobin Shen, Yongguang Zhang, and Qun Li. Dozyap: power-efficient wi-fi tethering. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 421–434. ACM, 2012. 106
- [77] Seungyeop Han, Haichen Shen, Matthai Philipose, Sharad Agarwal, Alec Wolman, and Arvind Krishnamurthy. Mcdnn: An approximation-based execution framework for deep stream processing under resource constraints. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 123–136. ACM, 2016. 105
- [78] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015. 105
- [79] 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. 4
- [80] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016. 34, 39, 73
- [81] João F Henriques, Rui Caseiro, Pedro Martins, and Jorge Batista. Exploiting the circulant structure of tracking-by-detection with kernels. In *European conference on computer vision*, pages 702–715. Springer, 2012. 82
- [82] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012. 4
- [83] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015. 105
- [84] Cristy Ho and Charles Spence. Verbal interface design: Do verbal directional cues auto-

- matically orient visual spatial attention? *Computers in Human Behavior*, 22(4):733–748, 2006. 48
- [85] Steve Hodges, Emma Berry, and Ken Wood. SenseCam: A wearable camera that stimulates and rehabilitates autobiographical memory. *Memory*, 19(7):685–696, 2011. 102
- [86] Berthold KP Horn and Brian G Schunck. Determining optical flow. *Artificial intelligence*, 17(1-3):185–203, 1981. 38
- [87] John A Hoxmeier and Chris DiCesare. System response time and user satisfaction: An experimental study of browser-based applications. *AMCIS 2000 Proceedings*, page 347, 2000. 104
- [88] Wenlu Hu, Brandon Amos, Zhuo Chen, Kiryong Ha, Wolfgang Richter, Padmanabhan Pillai, Benjamin Gilbert, Jan Harkes, and Mahadev Satyanarayanan. The case for offload shaping. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*, pages 51–56. ACM, 2015. 106
- [89] Feng-Cheng Huang, Shi-Yu Huang, Ji-Wei Ker, and Yung-Chang Chen. High-performance sift hardware accelerator for real-time image feature extraction. *IEEE Transactions on Circuits and Systems for Video Technology*, 22(3):340–351, 2012. 104
- [90] Huawei. Kirin. <https://en.wikipedia.org/wiki/HiSilicon>. Accessed on October 6, 2017. 104
- [91] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Quantized neural networks: Training neural networks with low precision weights and activations. *arXiv preprint arXiv:1609.07061*, 2016. 105
- [92] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016. 105
- [93] Emmanuel Iarussi, Adrien Bousseau, and Theophanis Tsandilas. The drawing assistant: Automated drawing guidance and feedback from photographs. In *ACM Symposium on User Interface Software and Technology (UIST)*. ACM, 2013. 42
- [94] Ikea. Skogsta Stool. <http://www.ikea.com/us/en/catalog/products/10297958/>, 2017. 46
- [95] Open Edge Computing Initiative. Living Edge Lab. <http://openedgecomputing.org/le1.pdf>. Accessed on October 6, 2017. 23, 54, 112
- [96] Intel. Movidius. <https://www.movidius.com/>. Accessed on October 6, 2017. 104
- [97] Puneet Jain, Justin Manweiler, and Romit Roy Choudhury. Overlay: Practical mobile augmented reality. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 331–344. ACM, 2015. 102
- [98] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014. 46

- [99] Zdenek Kalal, Krystian Mikolajczyk, and Jiri Matas. Forward-backward error: Automatic detection of tracking failures. In *Pattern recognition (ICPR), 2010 20th international conference on*, pages 2756–2759. IEEE, 2010. 82
- [100] Zdenek Kalal, Krystian Mikolajczyk, and Jiri Matas. Tracking-learning-detection. *IEEE transactions on pattern analysis and machine intelligence*, 34(7):1409–1422, 2012. 82
- [101] Vaiva Kalnikaite, Abigail Sellen, Steve Whittaker, and David Kirk. Now let me see where i was: understanding how lifelogs mediate memory. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2045–2054. ACM, 2010. 102
- [102] Michal Kampf, Israel Nachson, and Harvey Babkoff. A serial test of the laterality of familiar face recognition. *Brain and cognition*, 50(1):35–50, 2002. 47
- [103] Yan Ke, Rahul Sukthankar, and Martial Hebert. Event detection in crowded videos. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8, 2007. 34, 39
- [104] Kyungnam Kim, Thanarat H Chalidabhongse, David Harwood, and Larry Davis. Real-time foreground–background segmentation using codebook model. *Real-time imaging*, 11(3):172–185, 2005. 39
- [105] Davis E King. Dlib-ml: A machine learning toolkit. *The Journal of Machine Learning Research*, 10:1755–1758, 2009. 39, 46
- [106] Roberta L Klatzky, Pnina Gershon, Vikas Shivaprabhu, Randy Lee, Bing Wu, George Stetten, and Robert H Swendsen. A model of motor performance during surface penetration: from physics to voluntary control. *Experimental brain research*, 230(2):251–260, 2013. 48, 51
- [107] Sokol Kosta, Andrius Aucinas, Pan Hui, Richard Mortier, and Xinwen Zhang. Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Infocom, 2012 Proceedings IEEE*, pages 945–953. IEEE, 2012. 103
- [108] Albert M Lai and Jason Nieh. On the performance of wide-area thin-client computing. *ACM Transactions on Computer Systems (TOCS)*, 24(2):175–209, 2006. 104
- [109] Yong Jae Lee, C. Lawrence Zitnick, and Michael F. Cohen. Shadowdraw: real-time user guidance for freehand drawing. *ACM Transactions on Graphics (TOG)*, 30(4), 2011. 42
- [110] Lego. Life of George. <http://george.lego.com/>, 2011. 40
- [111] Michael B Lewis and Andrew J Edmonds. Face detection: Mapping human performance. *Perception*, 32(8):903–920, 2003. 10
- [112] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 1–14. ACM, 2010. 8, 14, 56
- [113] Robert LiKamWa and Lin Zhong. Starfish: Efficient concurrency support for computer vision applications. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, pages 213–226. ACM, 2015. 106
- [114] Robert LiKamWa, Bodhi Priyantha, Matthai Philipose, Lin Zhong, and Paramvir Bahl.

- Energy characterization and optimization of image sensing toward continuous mobile vision. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 69–82. ACM, 2013. 106
- [115] Robert LiKamWa, Zhen Wang, Aaron Carroll, Felix Xiaozhu Lin, and Lin Zhong. Draining our glass: An energy and heat characterization of google glass. In *Proceedings of 5th Asia-Pacific Workshop on Systems*, page 10. ACM, 2014. 85, 106
- [116] Robert LiKamWa, Yunhui Hou, Julian Gao, Mia Polansky, and Lin Zhong. Redeye: analog convnet image sensor architecture for continuous mobile vision. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 255–266. IEEE Press, 2016. 106
- [117] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. Learning efficient convolutional networks through network slimming. *arXiv preprint arXiv:1708.06519*, 2017. 105
- [118] Jack M Loomis, Reginald G Golledge, and Roberta L Klatzky. Navigation system for the blind: Auditory display modes and guidance. *Presence: Teleoperators and Virtual Environments*, 7(2):193–203, 1998. 1
- [119] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004. 30, 34, 45, 104
- [120] Chaochao Lu and Xiaoou Tang. Surpassing human-level face verification performance on lfw with gaussianface. In *AAAI*, pages 3811–3819, 2015. 4
- [121] Manuel Martinez, Alvaro Collet, and Siddhartha S Srinivasa. Moped: A scalable and low latency object recognition and pose estimation system. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 2043–2049. IEEE, 2010. 30
- [122] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014. 20
- [123] Microsoft. Microsoft Hololens. <https://www.microsoft.com/en-us/hololens>, 2017. 18
- [124] Microsoft. Cortana. <https://www.microsoft.com/en-us/windows/cortana>. Accessed on October 5, 2017. 102
- [125] Robert B Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 267–277. ACM, 1968. 48, 49, 104
- [126] Martin Missfeldt. How Google Glass Works. <http://www.brillen-sehhilfen.de/en/googleglass/>, 2013. 5
- [127] Myoonet. Myoonet private cloud. <http://www.myoonet.com/theMyoonet.html>, 2017. 4
- [128] Fiona Fui-Hoon Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour & Information Technology*, 23(3):153–163, 2004. 104
- [129] Dushyanth Narayanan and Mahadev Satyanarayanan. Predictive resource management

- for wearable computing. In *Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 113–128. ACM, 2003. 12
- [130] Shahriar Nirjon, Robert F Dickerson, Qiang Li, Philip Asare, John A Stankovic, Dezhi Hong, Ben Zhang, Xiaofan Jiang, Guobin Shen, and Feng Zhao. Musicalheart: A hearty way of listening to music. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, pages 43–56. ACM, 2012. 12
- [131] Shahriar Nirjon, Robert Dickerson, John Stankovic, Guobin Shen, and Xiaofan Jiang. sm-fcc: exploiting sparseness in speech for fast acoustic feature extraction on mobile devices—a feasibility study. In *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*, page 8. ACM, 2013. 89
- [132] Brian D Noble and Mahadev Satyanarayanan. Experience with adaptive mobile applications in odyssey. *Mobile Networks and Applications*, 4(4):245–254, 1999. 103
- [133] Brian D Noble, Mahadev Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R Walker. Agile application-aware adaptation for mobility. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 276–287. ACM, 1997. 12
- [134] Nokia. Multi-access edge computing. <https://networks.nokia.com/solutions/multi-access-edge-computing>, 2017. 4
- [135] ODG. ODG R7 smartglass. <https://shop.osterhoutgroup.com/products/r-7-glasses-system>, 2017. 5, 18
- [136] OpenStack. <http://www.openstack.org/>, February 2015. 26, 52
- [137] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *Knowledge and Data Engineering, IEEE Transactions on*, 22(10):1345–1359, 2010. 46
- [138] Raja Parasuraman and D Roy Davies. Decision theory analysis of response latencies in vigilance. *Journal of Experimental Psychology: Human Perception and Performance*, 2(4):578, 1976. 47
- [139] Donald J Patterson, Lin Liao, Krzysztof Gajos, Michael Collier, Nik Livic, Katherine Olson, Shiaokai Wang, Dieter Fox, and Henry Kautz. Opportunity knocks: A system to provide cognitive assistance with transportation services. In *UbiComp 2004: Ubiquitous Computing*, pages 433–450. Springer, 2004. 102
- [140] Andriy Pavlovyh and Wolfgang Stuerzlinger. Target following performance in the presence of latency, jitter, and signal dropouts. In *Proceedings of Graphics Interface 2011*, pages 33–40. Canadian Human-Computer Communications Society, 2011. 84, 104
- [141] Michael I Posner and Steven E Petersen. The attention system of the human brain. Technical report, DTIC Document, 1989. 13
- [142] Moo-Ryong Ra, Anmol Sheth, Lily Mummert, Padmanabhan Pillai, David Wetherall, and Ramesh Govindan. Odessa: enabling interactive perception applications on mobile devices. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 43–56. ACM, 2011. 103
- [143] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism,

- locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, 48(6):519–530, 2013. 105
- [144] Meike Ramon, Stephanie Caharel, and Bruno Rossion. The speed of recognition of personally familiar faces. *Perception*, 40(4):437–449, 2011. 10, 47
- [145] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnet: Imagenet classification using binary convolutional neural networks. *arXiv preprint arXiv:1603.05279*, 2016. 63
- [146] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015. 4, 34, 46, 67
- [147] Gavriel Salvendy. *Handbook of human factors and ergonomics*. John Wiley & Sons, 2012. 47
- [148] Mahadev Satyanarayanan. From the editor in chief: Augmenting cognition. *IEEE Pervasive Computing*, 3(2):4–5, 2004. 4
- [149] Mahadev Satyanarayanan and Dushyanth Narayanan. Multi-fidelity algorithms for interactive mobile applications. *Wireless Networks*, 7(6):601–607, 2001. 103
- [150] 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. 4, 14
- [151] Michel Schmitt, Albert Postma, and Edward De Haan. Interactions between exogenous auditory and visual spatial attention. *The Quarterly Journal of Experimental Psychology: Section A*, 53(1):105–130, 2000. 48
- [152] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 815–823, 2015. 4, 39
- [153] Longfei Shangguan, Zheng Yang, Zimu Zhou, Xiaolong Zheng, Chenshu Wu, and Yunhao Liu. Crossnavi: enabling real-time crossroad navigation for the blind with commodity phones. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 787–798. ACM, 2014. 102
- [154] Donghwa Shin, Younghyun Kim, Naehyuck Chang, and Massoud Pedram. Dynamic voltage scaling of oled displays. In *Design Automation Conference (DAC), 2011 48th ACM/EDAC/IEEE*, pages 53–58. IEEE, 2011. 85, 106
- [155] Ben Shneiderman and Catherine Plaisant. *Designing the user interface (4th edition)*. Pearson Addison Wesley, USA, 1987. 49
- [156] Jamie Shotton, Matthew Johnson, and Roberto Cipolla. Semantic texton forests for image categorization and segmentation. In *Computer vision and pattern recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008. 30
- [157] Daniel Siewiorek, Asim Smailagic, and Anind Dey. Architecture and applications of virtual coaches. *Quality of Life Technology Handbook*, page 197, 2012. 101

- [158] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 79
- [159] Small World Toys. Toy Country Club Sandwich. <http://www.smallworldtoys.com/index.php/categories/small-world-living/country-club-sandwich.html>. 45
- [160] Ray Smith. An overview of the tesseract ocr engine. In *icdar*, pages 629–633. IEEE, 2007. 10, 13, 30
- [161] Jan Sochman and Jiri Matas. Waldboost-learning for time constrained sequential detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 150–156. IEEE, 2005. 105
- [162] Heikki Summala. Brake reaction times and driver behavior analysis. *Transportation Human Factors*, 2(3):217–226, 2000. 47
- [163] Ivan E Sutherland. A head-mounted three dimensional display. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 757–764. ACM, 1968. 5
- [164] Yaniv Taigman, Ming Yang, Marc’Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1701–1708, 2014. 4, 39
- [165] Gabriel Takacs, Maha El Choubassi, Yi Wu, and Igor Kozintsev. 3d mobile augmented reality in urban scenes. In *Multimedia and Expo (ICME), 2011 IEEE International Conference on*, pages 1–4. IEEE, 2011. 30, 31, 102
- [166] Titus JJ Tang and Wai Ho Li. An assistive eyewear prototype that interactively converts 3d object locations into spatial audio. In *Proceedings of the 2014 ACM International Symposium on Wearable Computers*, pages 119–126. ACM, 2014. 34, 38
- [167] Enrico Tanuwidjaja, Derek Huynh, Kirsten Koa, Calvin Nguyen, Churen Shao, Patrick Torbett, Colleen Emmenegger, and Nadir Weibel. Chroma: a wearable augmented-reality solution for color blindness. In *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 799–810. ACM, 2014. 1, 102
- [168] Michael J Tarr, Daniel Kersten, and Heinrich H Bülthoff. Why the visual recognition system might encode the effects of illumination. *Vision research*, 38(15):2259–2275, 1998. 47
- [169] Brandon Taylor, Yoshihisa Abe, Anind Dey, Mahadev Satyanarayanan, Dan Siewiorek, and Asim Smailagic. Virtual machines for remote computing: Measuring the user experience, 2015. 104
- [170] Niraj Tolia, David G Andersen, and Mahadev Satyanarayanan. Quantifying interactive user experience on thin clients. *Computer*, 39(3):46–52, 2006. 104
- [171] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of cognitive neuroscience*, 3(1):71–86, 1991. 13, 29, 30, 73
- [172] Unity. Unity Game Engine. <https://unity3d.com/>, 2017. 19

- [173] Nuno Vasconcelos and Mohammad J Saberian. Boosting classifier cascades. In *Advances in Neural Information Processing Systems*, pages 2047–2055, 2010. 105
- [174] VeryPDF. PDF to Text OCR Converter Command Line. <http://www.verypdf.com/app/pdf-to-text-ocr-converter/index.html>, 2017. 30
- [175] 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–511. IEEE, 2001. 74, 105
- [176] Quest Visual. Word Lens. https://en.wikipedia.org/wiki/Word_Lens, 2010. 102
- [177] Vuzix. Vuzix M100 smartglass. <https://www.vuzix.com/Products/M100-Smart-Glasses>, 2017. 18
- [178] Junjue Wang. Tool for Painless Object Detector. <https://github.com/junjuew/TPOD>, 2017. Accessed: 2017-09-10. 82
- [179] Yi Wang, Jialiu Lin, Murali Annamaram, Quinn A Jacobson, Jason Hong, Bhaskar Krishnamachari, and Norman Sadeh. A framework of energy efficient mobile sensing for automatic user state recognition. In *Proceedings of the 7th international conference on Mobile systems, applications, and services*, pages 179–192. ACM, 2009. 85, 106
- [180] Benjamin Watson, Victoria Spaulding, Neff Walker, and William Ribarsky. Evaluation of the effects of frame time variation on vr task performance. In *Virtual Reality Annual International Symposium, 1997., IEEE 1997*, pages 38–44. IEEE, 1997. 84, 104
- [181] Xilinx. Xilinx. <https://en.wikipedia.org/wiki/Xilinx>. 104
- [182] Fengyuan Xu, Yunxin Liu, Thomas Moscibroda, Ranveer Chandra, Long Jin, Yongguang Zhang, and Qun Li. Optimizing background email sync on smartphones. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 55–68. ACM, 2013. 106
- [183] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014. 79
- [184] Bin Zhang, Weibei Dou, and Liming Chen. Ball hit detection in table tennis games based on audio analysis. In *Pattern Recognition, 2006. ICPR 2006. 18th International Conference on*, volume 3, pages 220–223. IEEE, 2006. 88
- [185] Tong Zhang and C-C Jay Kuo. Audio content analysis for online audiovisual data segmentation and classification. *IEEE Transactions on speech and audio processing*, 9(4): 441–457, 2001. 88
- [186] Fang Zheng, Guoliang Zhang, and Zhanjiang Song. Comparison of different implementations of mfcc. *Journal of Computer Science and Technology*, 16(6):582–589, 2001. 88
- [187] Xizhou Zhu, Yuwen Xiong, Jifeng Dai, Lu Yuan, and Yichen Wei. Deep feature flow for video recognition. *arXiv preprint arXiv:1611.07715*, 2016. 105