

# **An Application Framework for Wearable Cognitive Assistance**

Zhuo Chen

[zhuoc@cs.cmu.edu](mailto:zhuoc@cs.cmu.edu)

Oct 24

## **THESIS PROPOSAL**

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 © 2016 Zhuo Chen

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

## **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 framework that simplifies the creation of and experimentation with this new genre of applications. An essential capability of this framework 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.

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Crisp Interactive Response . . . . .	2
1.2	Limitation of Mobile Devices . . . . .	3
1.3	Coarse-grain Parallelism . . . . .	4
1.4	Thesis Statement . . . . .	4
<b>2</b>	<b>Gabriel Architecture</b>	<b>6</b>
2.1	Design and Implementation . . . . .	6
2.1.1	Overview . . . . .	6
2.1.2	Use of Cloudlets . . . . .	7
2.1.3	Limiting Queueing Delay . . . . .	8
2.1.4	Implementation Details . . . . .	9
2.2	Micro Benchmarks . . . . .	10
2.2.1	Gabriel Overhead . . . . .	10
2.2.2	Queueing Delay Mitigation . . . . .	11
<b>3</b>	<b>Applications</b>	<b>13</b>
3.1	Application Implementation . . . . .	13
3.2	Quantifying Latency Bounds . . . . .	13
3.2.1	Relevant bounds in previous studies . . . . .	15
3.2.2	Deriving bounds from physical motion . . . . .	16
3.2.3	Bounds for step-by-step instructed tasks . . . . .	16
3.3	Application Performance . . . . .	19
3.3.1	Experimental Setup . . . . .	19
3.3.2	Response Time with WiFi Cloudlet Offloading . . . . .	20
3.3.3	Benefits of Cloudlets . . . . .	21
<b>4</b>	<b>Performance Optimization with Coarse-grain Parallelism</b>	<b>23</b>
4.1	Improving System Throughput . . . . .	24
4.1.1	Throughput vs. User Experienced Response Time . . . . .	24
4.1.2	Experiment . . . . .	25
4.2	Improving System Latency . . . . .	25
4.2.1	Background . . . . .	25
4.2.2	Approach . . . . .	26

4.2.3	Early Results . . . . .	27
<b>5</b>	<b>Path to Completion</b>	<b>30</b>
5.1	Extend Gabriel with New Applications . . . . .	30
5.2	Improve System Latency . . . . .	31
5.3	Optimization on Energy Consumption . . . . .	31
5.3.1	Motivation . . . . .	31
5.3.2	Approach . . . . .	32
5.4	Topics Not Covered . . . . .	32
5.5	Timeline . . . . .	33
<b>6</b>	<b>Related Work</b>	<b>34</b>
6.1	Wearable Cognitive Assistance . . . . .	34
6.2	Latency Sensitivity of Mobile Applications . . . . .	34
6.3	Cyber Foraging . . . . .	35
6.4	Speeding up Computer Vision . . . . .	35
	<b>Bibliography</b>	<b>37</b>

# 1 Introduction

Using a mobile device to provide a user with cognitive assistance is one of the early goals in mobile computing. As early as 1998, Loomis et al. [35] demonstrated the use of a laptop for providing walking directions for the blind. In the same year, Davies et al. [13] 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 [58] 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 algorithm 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 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 crisp response. This thesis targets at enabling a new genre of cognitive assistance applications that can leverage heavy computation at very low system latency.

Figure 1.1 presents some hypothetical use cases of such 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 error 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 too heavy to be run on a mobile device, which is constrained by its weight, size, and thermal dissipation.

The core contribution of my thesis is the design, implementation, and evaluation of Gabriel, an application framework that simplifies the creation and experimentation of new wearable cognitive assistance applications. This framework factors out the common functionalities 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. It also provides mechanisms to carefully control the energy consumption on a mobile device. So far, I have built seven applications using Gabriel and have confirmed the performance, generality, and ease of use of Gabriel as a “plug-and-play” platform.

One key factor 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 minimum jitter for a connection to the mobile user, making it the ideal place for offload-

Jane wants to make her first butterscotch pudding today. She starts her “Recipe” application in her Google 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 half a cup, since this is all she has at home, hoping this is not a critical step. This is quickly caught by the Glass, which then yells “This is not enough. Please go and buy more.”

(a) Cooking

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” Google Glass application to guide him in the assembly steps. Bob feels it’s much easier to read instructions from Glass and finishes everything ahead of schedule. When he tries to sit on the chair he last assembled, the Glass warns him that he didn’t screw the last nail tight enough...

(b) Furniture Assembly

Figure 1.1: Task Assistance Scenarios

ing. Recent work in both research community and industry have pushed cloudlets to real world deployment [9].

The rest of this document goes as follows. The remainder of this chapter explains the key design consideration of the Gabriel framework, followed by the thesis statement. In Chapter 2, Gabriel is explained in detail. Chapter 3 presents a set of wearable cognitive assistance applications that I have built on top of Gabriel, which are used to evaluate the system performance. Chapter 4 explains how the performance can be improved through coarse-grain parallelism. Chapter 5 shows the plan to complete the thesis, and Chapter 6 discusses related work.

## 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 an application that does face recognition for a user, the recognized person in the scene may have already left before slow guidance regarding the person’s information arrives). Even if a late guidance still provides useful information, the user may be annoyed by a long delay.

How fast is fast enough? Although to answer this question requires careful study of the user and task, it is best if our system can match or even perform better than the speed of a human on the task. In the example of face detection and recognition, Lewis et al. [32] 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. [46] 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. [3] report that human subjects recognize short target phrases within 300 to 450 ms. If the Gabriel-based applications want to meet these numbers, the latency overhead of the system infrastructure could only be a few tens of millisecond, leaving enough time for task specific processing.

Year	Processor	Typical Server Speed	Device	Typical Handheld or Wearable 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® E5-2699v4	96.8 GHz (2x22 cores)	Samsung Galaxy S7	7.5 GHz (4 cores)

Table 1.1: Evolution of Hardware Performance (Source: adapted from Flinn [14])

## 1.2 Limitation of Mobile Devices

Mobile devices are always resource poor relative to fixed infrastructure in computing power. Figure 1.1, adapted from Flinn [14], illustrates the consistent large gap in the processing power of typical server and mobile device hardware over a near 20-year period. 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 better. 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 improve the processing power of a mobile device.

The large gap in the processing capabilities of wearable and server hardware directly determines the 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 [54]). When the entire application is running on a Google Glass, 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) through WiFi 802.11n, it takes 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 on a mobile device. It is a more serious problem for wearable devices, as the limited size and weight incurs strict constraints on the amount of battery that can be embedded. For example, a Google Glass’s 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 natively. For the reasons above, the Gabriel framework must be built to leverage server resources for faster computation and efficient energy usage.

### 1.3 Coarse-grain Parallelism

In real life, people may need assistance in multiple cognitive tasks at the same time. For example, a user just entering a new building may need a navigation assistant to get to the room she wants, an augmented reality assistant to display relevant information about important objects around her, and a face recognition assistant to help her in social interactions. Each of these cognitive tasks may also be built on top of multiple independent building blocks. The navigation assistant, for example, can use the WiFi and IMU sensor data for dead reckoning, while leveraging image sensing for periodic calibration. These different tasks and software building blocks are natural units for coarse-grain parallelism.

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 [62], natural language translation [6], and OCR [54], they are usually written in a variety of programming languages and use diverse runtime systems. In their entirety, these existing 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 the 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 section 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, sense the tone in which they are spoken, interpret the facial expressions with which they are spoken, and observe the body language and gestures that accompany them. There is substantial evidence [43] 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.

### 1.4 Thesis Statement

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

I plan to validate the thesis statement from the following aspects.

- *Design and implement a prototype of the Gabriel framework.* This application framework provides mechanisms to simplify the exploitation of coarse-grain parallelism and energy conservation for wearable cognitive assistance applications. Common functionalities of cognitive assistance applications are implemented in Gabriel and will be validated using micro benchmarks.
- *Build cognitive assistance applications on top of Gabriel.* These applications cover a variety of cognitive assistance tasks, use different mobile hardware, and leverage diverse computer vision and signal processing techniques. They can serve as a benchmark for thorough system analysis.
- *Evaluate the system performance of the applications under different system conditions.* By experimentally studying the latency contributions of different application and system layers, I will show the benefits of using cloudlets, as well as other infrastructure needed to support these applications.
- *Demonstrate how the coarse-grain parallelism in Gabriel can improve system performance.* Using this approach, a subset of the benchmark applications will be restructured to improve both throughput and latency.
- *Show how Gabriel can be leveraged to reduce energy consumption on mobile devices.* A set of standard APIs will be designed to control the mobile sensors. The tradeoff between system performance and energy will be explored.

By doing all the validations above, I will demonstrate the possibility of building computation heavy and latency sensitive cognitive assistance applications using today's mobile and server hardware.

## 2 Gabriel Architecture

### 2.1 Design and Implementation

#### 2.1.1 Overview

I have designed and implemented the prototype of Gabriel, a software framework for developing wearable cognitive assistance applications. Gabriel can be viewed as a PaaS (Platform as a Service) layer, which factors out complex system-level functionality that is common across many applications. It is an open-source project at <https://github.com/cmusatyalab/gabriel>.

Figure 2.1 illustrates 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 functionalities 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. In my implementation, the PubSub mechanism is achieved by starting a TCP server and service registry in the Control VM. Each Cognitive VM registers to the Control VM for the streams it wants and receives the data from the TCP channel.

The outputs of the Cognitive VMs are sent to a single *User Guidance VM* that integrates these outputs and performs higher-level cognitive processing. From time to time, this triggers output for user assistance and transmits the guidance back to the wearable device. The guidance can be in the format of a spoken audio whispered into the user’s ears, an image displayed on the screen of the wearable device, an animation consisting multiple images, or containing multiple of the above. The guidance is sent using a JSON structure.

The outputs of the cognitive VMs can also be used to infer a user’s context for sensor control of the wearable device. This can be a useful technique to save energy on the device. For example, if a user is taking a break sitting in her chair in the middle of furniture assembly, her Glass device does not have to capture video and stream it for assembly related cognitive assistance. This kind of sensor control is achieved in Gabriel through a context inference module in the Control VM. It sends control messages back to the wearable device to turn on/off a sensor or adjust data capture rate, triggered by a user’s context change.

While the PubSub backbone provides a clear separation between the computation in Cognitive VMs and that in the User Guidance VM, Gabriel doesn’t have any restriction on how to separate the computation. For example, each Cognitive VM can be an independent application

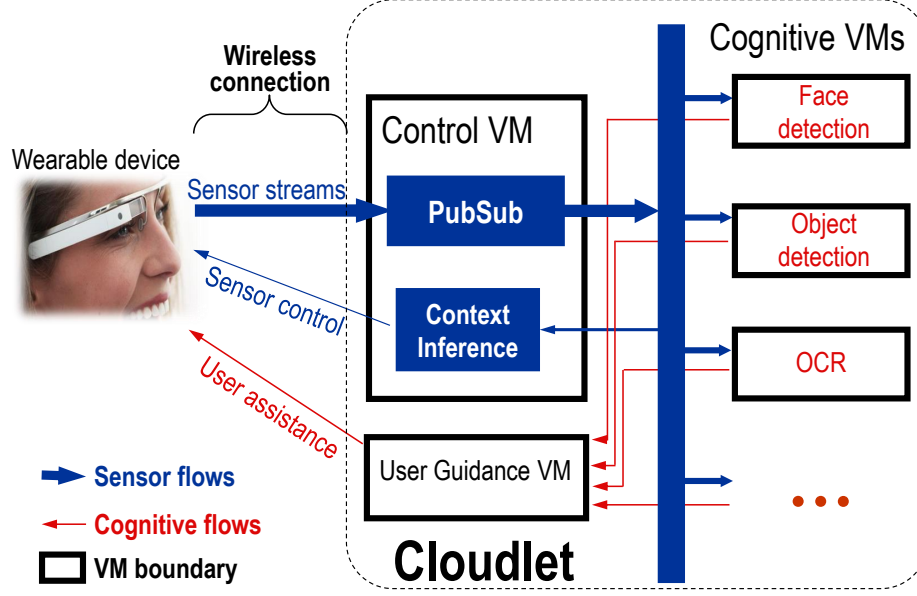


Figure 2.1: Gabriel Architecture

for some task, such as navigation, face recognition, or language translation, while the User Guidance VM simply forwards guidance from each application to the user. The Cognitive VMs can also be components within the same application (e.g. object detector and human action detector in an assembly task), in which case the User Guidance VM has to do task related processing based on the lower level detection results from all Cognitive VMs. The flexibility of such architecture enables quick prototyping of new applications and ease of experimentation. To handle arbitrary task graphs in which task components are organized differently from a two-level hierarchy, or to handle multiple tasks each consisting of multiple components, the Cognitive VMs can also register their results to the PubSub backbone to be leveraged by other Cognitive VMs, although this approach is currently not supported.

### 2.1.2 Use of Cloudlets

Gabriel faces the difficult problem of simultaneously satisfying the need for crisp, low-latency interaction (Section 1.1) and the need for offloading computation from a wearable device (Section 1.2). The obvious solution of using commercial cloud services over a WAN is unsatisfactory because the RTT is too long. Li et al. [33] 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 1.1.

Gabriel achieves low-latency processing by offloading to cloudlets [48]. A cloudlet is a powerful, well-connected small datacenter that is just one wireless hop away. It can be viewed 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 attentions from industry for real deployments, and they may be

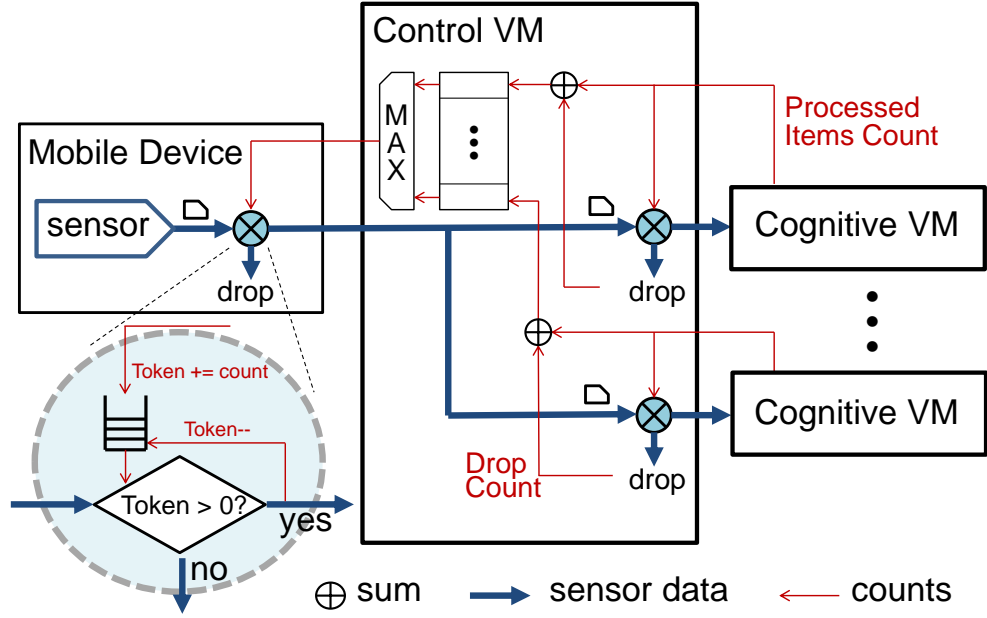


Figure 2.2: Two Level Token-based Filtering Scheme

variously known as micro data centers [17], fog [7], or mobile edge cloud [9]. We use the term “cloudlet” throughout the document to refer to such 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 [19]. 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.

### 2.1.3 Limiting Queueing 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 2.2). 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.

Implemented as part of Gabriel, this token-based flow control mechanism is a general approach that could help a system to get rid of system queueing delay, which is especially helpful for latency-sensitive applications. It also simplifies dealing with multiple processing components with different throughput. The token number provides a simple interface to trade latency for throughput.

## **2.1.4 Implementation Details**

### **Mobile front-end**

The mobile front-end is implemented in Android 4.4.2, since our initial target mobile device is Google Glass. The Glass 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 currently Google Glass only supports WiFi due to its hardware limitation. Once a guidance message is received, the Glass displays visual outputs as bitmap images on the screen and audio guidance as synthesized speech using the Android text-to-speech API.

The portability of Android makes it easy to run the Gabriel front-end software on compatible smartphones without any change in code. 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 of these applications to other wearable devices with different development platform, such as iOS and Windows. On the other hand, one can add additional computation in client software to do simple preprocessing of sensor data to reduce network bandwidth usage and energy consumption. For example, the Offload Shaping work [24] reports a few case studies in different application scenarios.

### **Virtual Machine Ensemble**

As explained earlier in Section 1.3, 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 framework, as long as it follows the communication protocol with the control VM.

The use of VM 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. Although recent work on using containers [36] has offered similar properties, they are restrictive in terms of OS choices. I restrict the attempt to use only VMs in this thesis.

## Discovery and Initialization

The Gabriel back-end consists of many software components in multiple VMs working together. Key to making this composition work is a mechanism for the different components to easily discover and connect with each other. In the prototype, 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. Each of the cognitive VMs registers itself to the control VM and subscribes to the desired sensor streams through the PubSub system. Their own processed data streams are also registered to the PubSub system to be received by the User Guidance VM.

## 2.2 Micro Benchmarks

### 2.2.1 Gabriel Overhead

Gabriel uses VMs extensively so that there are few constraints on the operating systems and cognitive engines 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 engine 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. Figure 2.1 summarizes the time spent in Gabriel 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.

The end-to-end latency measures at the Glass device are shown in Figure 2.3. These include processing time to capture images on Glass, time to send 6–67 KB images over the Wi-Fi network, the Gabriel response times with the *NULL* engine, and transmission of dummy results back over Wi-Fi. I also compare against an ideal server – essentially the *NULL* server running natively on the offload machine (no VM or Gabriel). Here, Gabriel and the ideal achieve 33 and 29 ms median response times respectively, confirming an approximately 4 ms overhead for the flexible Gabriel architecture. These end-to-end latencies represent the minimum achievable for offload of a trivial task. Real cognitive computations will add to these values.

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

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

Table 2.1: Intrinsic Delay Introduced by Gabriel

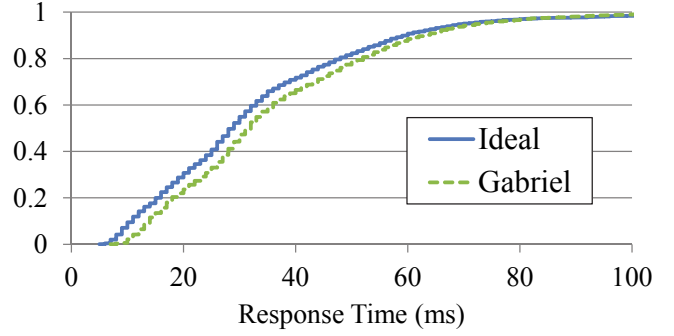
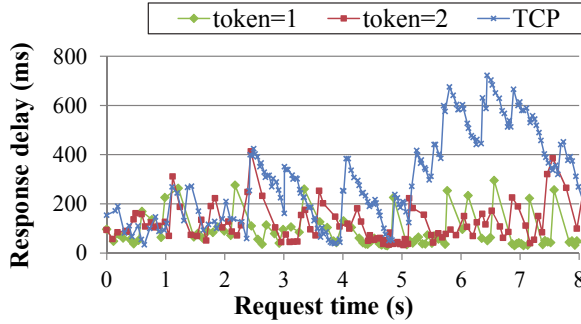
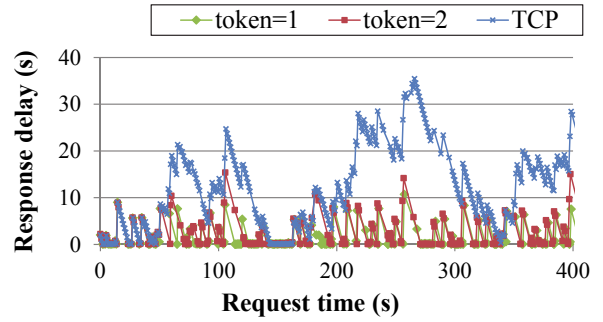


Figure 2.3: CDF of End-to-end Response Time at Glass Device for the *NULL* Cognitive Engine



(a) Synthetic Cognitive Engine (20 FPS ingress)



(b) OCR (1 FPS ingress)

Figure 2.4: Response Delay of Request

### 2.2.2 Queueing Delay Mitigation

In order to evaluate the need for the application-level, end-to-end flow-control mechanism for mitigating queueing, 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 queueing in check. Unfortunately, the processing times of cognitive engines are highly variable and content-dependent. An image that takes much longer than average to process will cause queueing and an increase in latency for subsequent ones. However, with our two-level token bucket mechanism, explicit feedback controls the input rate to avoid building up queues.

I first compare the differences in queueing behavior with and without our flow control mechanism using a synthetic cognitive engine. 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 2.4(a), without application-level flow control (labeled TCP), many frames exhibit much higher latencies than expected due to queueing. In contrast, the traces using our token-based flow control mecha-



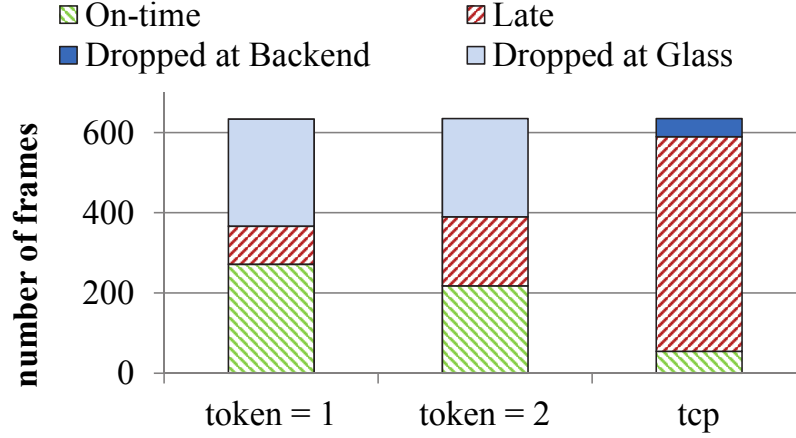


Figure 2.5: Breakdown of Processed and Dropped Frame for OCR (500ms for on-time response)

nism 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 engine. Here, the mean processing time is just under 1 second, so I use an offered load of 1 FPS. The traces in Figure 2.4(b) show similar results: latency can increase significantly when explicit flow control is not used (TCP), but remains under control with our 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 2.5 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 s 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 our 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 our token-based flow control mechanism is effective and necessary for reducing system latency.

## 3 Applications

### 3.1 Application Implementation

Using the Gabriel framework introduced in the previous section, I have built cognitive assistance applications for the seven tasks summarized in Table 3.1. Video demos of some applications are available at <http://goo.gl/02m0nL>. The diversity of these applications makes them a good benchmark suite for analyzing system performance. Among these tasks, some require prompt response to a real-world event, such as ball motion in Ping-pong Assistant. Others, such as Sandwich Assistant, offer step-by-step guidance towards a pre-defined goal and have less demanding speed requirements. The use of computer vision varies considerably across these applications, spanning color and edge detection, motion analysis, and object recognition based on deep convolutional neural networks (DNNs).

On the Gabriel back-end, the workflow of all applications consists of two phases. 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 Assistant, 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.

### 3.2 Quantifying Latency Bounds

As stated in Section 1.1, all of the above applications have to provide guidance quickly enough so that the users won’t get annoyed. While it is easy to measure the system latency of the ap-







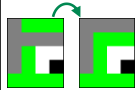





App Name	Example Input Video Frame	Description	Symbolic Rep	Guidance
Face		Jogs your memory on a familiar face whose name you cannot recall. Detects and extracts a tightly-cropped image of each face, and then applies a popular open-source face recognizer called OpenFace [3] that is based on a deep convolutional neural network (DNN) algorithm. Whispers name of person. Can be used in combination with <i>Expression</i> [4] to offer conversational hints.	ASCII text of name	Whispers "Barack Obama"
Pool		Helps a novice pool player aim correctly. Gives continuous visual feedback (left arrow, right arrow, or thumbs up) as the user turns his cue stick. Correct shot angle is calculated based on widely used <i>fractional aiming system</i> [1]. Color, line, contour, and shape detection are used. The symbolic rep describes the positions of the cue ball, object ball, target pocket, and the top and bottom of the cue stick.	<Pocket, object ball, cue ball, cue top, cue bottom>	
Ping-pong		Tells novice to hit ball to the left or right, depending on which is more likely to beat opponent. Uses color, line and optical-flow based motion detection to detect ball, table, and opponent. The symbolic rep is a 3-tuple: in rally or not, opponent position, ball position. Whispers "left" or "right". Alternatively it can offer spatial audio guidance using [45].	<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 [22] on a 10-15 frame video segment to classify the exercise. A poorly-performed repetition is classified as a distinct type of exercise (e.g. "good pushup" versus "bad pushup"). Uses smart phone on the floor for third-person viewpoint.	<Action count>	Says "8"
Lego		Guides a user in assembling 2D Lego models. Each video frame is analyzed in three steps: (i) finding the board using its distinctive color and black dot pattern; (ii) locating the Lego bricks on the board using edge and color detection; (iii) assigning brick color using weighted majority voting within each block. The symbolic rep is a matrix representing color for each brick.	[[0, 2, 1, 1], [0, 2, 1, 6], [2, 2, 2, 2]]	 Says "Find a 1x3 green piece and put it on top"
Draw		Helps a user to sketch better. Builds on third-party app [19] that was originally designed to take input sketches from pen-tablets and to output corrective guidance on a desktop screen. Our implementation preserves the back-end logic. A new Glass-based front-end allows a user to use any drawing surface and instrument and displays guidance on Glass. Displays the error alignment in sketch.		
Sandwich		Helps a cooking novice prepare sandwiches according to a recipe. Since real food is perishable, we use a children's food toy with realistic plastic ingredients. Object detection follows the RCNN deep neural net approach introduced in [11]. Implementation is on top of Caffe [20] and Dlib [23]. <i>Transfer learning</i> [33] helped us save time in labeling data and in training.	Object: "Lettuce on top of ham and bread"	 Says "Now put a piece of bread on the lettuce"

Table 3.1: Google Glass Cognitive Assistants Used in the Benchmark Suite

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

Figure 3.1: Latency Bounds of Assistant Applications

plications, it is difficult to know if they are fast enough without a reference target. Therefore, 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) [41], and environmental conditions (e.g. lighting) [59], 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 time-tested research (e.g., [8, 47]) 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 our implemented applications. This approach has the advantage of direct relevance, but may be limited in generality [56].

Table 3.1 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. [46] 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. [27], found normal humans take about 1000 ms. These studies suggest loose and tight bounds for Face Assistant that operates at par with human speeds.

Pool Assistant provides 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. [37] indicate that users can tolerate up to 100 ms

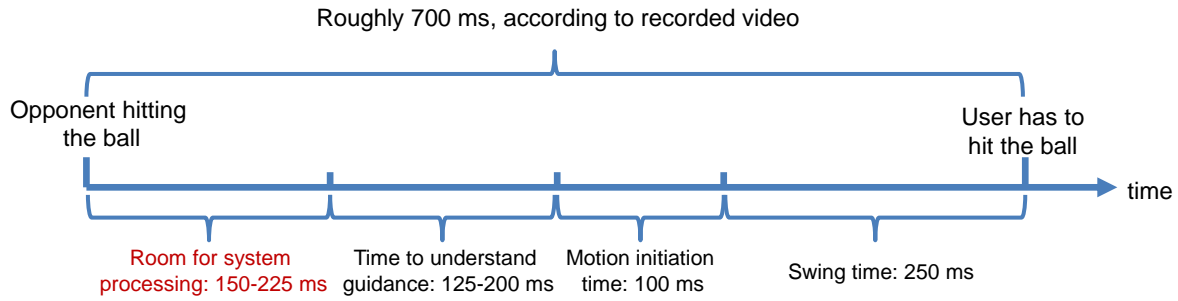


Figure 3.2: Deriving Latency Bound for Ping-pong Assistant from Physical Motion

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  $100 \pm 5$  ms should be applied to Pool application.

### 3.2.2 Deriving bounds from physical motion

Ping-pong Assistant has a clear latency bound defined by the moving speed of the Ping-pong ball. Figure 3.2 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 [21] 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 onset [49]. In either case, we need to allow 100 ms for initiation of motion [30]. 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 [8]. 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 bound of Work-out Assistant. 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 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, Drawing, and Sandwich Assistant 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 [37] 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

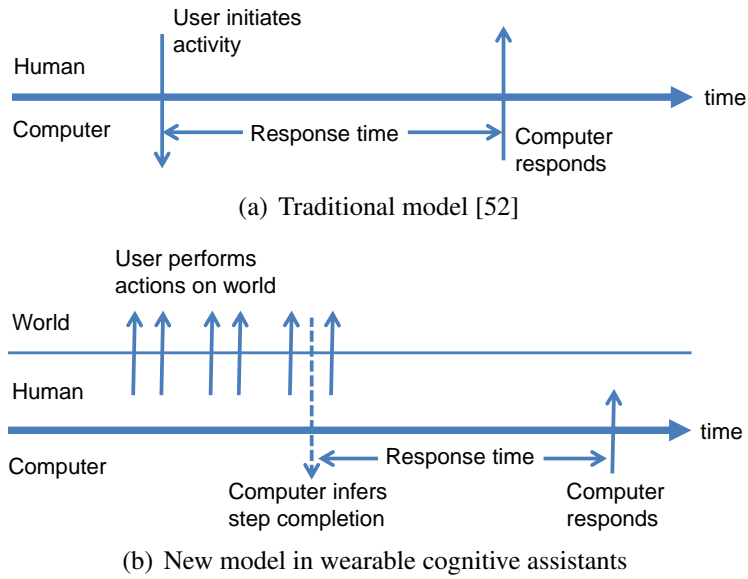


Figure 3.3: Simple Model of System Response Time and User Action/Think Time

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

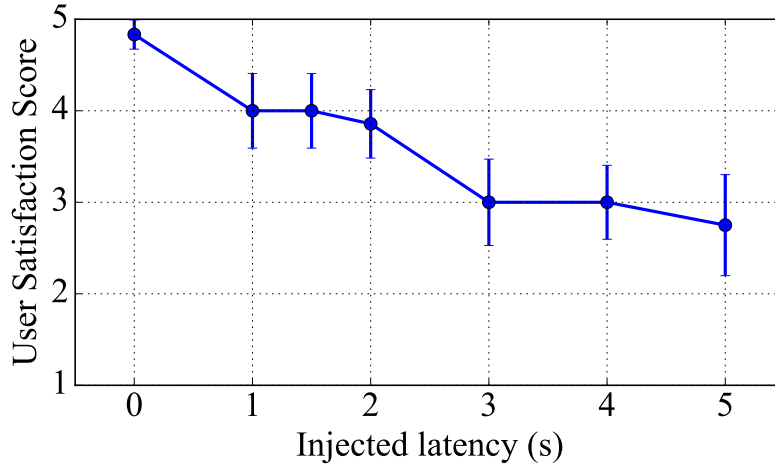
Table 3.2: Demographic Statistics

the interaction models of traditionally studied systems and our cognitive assistance applications (see Figure 3.3). 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, he manipulates the surroundings 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 he 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 3.2. 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



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

Figure 3.4: User Tolerance of System Latency

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 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 3.4 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

	CPU/GPU	RAM
Cloudlet	Intel <sup>®</sup> Core™ i7-3770, 3.4GHz, 4 cores, 8 threads	15GB
Cloudlet (GPU)	Intel <sup>®</sup> Xeon <sup>®</sup> E5-1630v3, 3.7GHz, 4 cores, 8 threads NVIDIA Tesla K40 (12GB RAM)	64GB
Cloud	Intel <sup>®</sup> Xeon <sup>®</sup> E5-2680v2, 2.8GHz, 8 VCPUs	15GB
Cloud (GPU)	Intel <sup>®</sup> Xeon <sup>®</sup> E5-2670, 2.6GHz, 8 VCPUs NVIDIA GRID K520 (4GB RAM)	15GB
Glass	TI OMAP4430, 1.2 GHz, 2 cores	773MB
Phone	Krait 450, 2.7 GHz, 4 cores	3GB

Table 3.3: Experiment Hardware Specifications

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 he started to signal for the next instruction to be around 700 ms on average. Allowing for motor initiation time of 100 ms [30], 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 three step-by-step instructed applications (Lego, Draw, and Sandwich Assistant).

## 3.3 Application Performance

In this section, I evaluate the performance of all the seven applications, in the context of meeting the latency bounds derived in the previous section. All of these applications will run on the Gabriel framework as independent Cognitive VMs. I will also vary some of the system parameters to see how the system performance changes.

### 3.3.1 Experimental Setup

The specifications of the hardware used in our experiments are shown in Table 3.3. Desktop-class machines running OpenStack [39] 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 same type of instance is used at the three different EC2 sites - East (Virginia), West (Oregon),



and Europe (Ireland). The Gabriel framework is run in VMs at both the cloud and the cloudlet. In all the experiments, our mobile devices are located on in the CMU campus.

For applications that use a GPU (e.g. Sandwich Assistant), 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 framework continues to run in a VM on OpenStack of the original cloudlet, connecting to the application service over local Ethernet.

I use Google Glass and Nexus 6 phone as mobile devices in our experiments. They connect to the cloudlet through a dedicated Wi-Fi access point to avoid interference. 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 Google Glass, I use ice packs to cool the device to avoid CPU clock rate fluctuation [20]. The Glass is also paired with a phone through Bluetooth, as suggested by [2], to get stable WiFi transmission in the experiments.

### 3.3.2 Response Time with WiFi Cloudlet Offloading

As most of the applications need a Google Glass to transmit first person video, it is used as the default mobile device for system measurement. According to the design of Gabriel framework, the Glass will offload computation to a cloudlet through WiFi. I measure the response times of all of the applications to see if they can meet the latency bounds derived in Section 3.2. Figure 3.5 shows the cumulative distribution functions (CDFs) of response times under various system setups, where the *solid black lines* correspond to the setup using Google Glass and WiFi cloudlets. Note that the applications only deliver guidance in response to a subset of the generated frames. The CDF graphs only include data associated with these frames.

As the figure shows, six of our seven applications successfully meet at least the loose latency bounds. All of the applications that provide step-by-step instructions easily meet the loose bounds. Lego and Sandwich Assistants can meet the tight bound as well, while Drawing Assistant comes close to achieving this. Face Assistant easily meets the loose latency bound, but not the tighter bound of 370 ms. Not surprisingly, the only application that fails to meet its bounds is Pool Assistant, which has the most stringent latency bound of 100 ms.

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, Glass exchanges timestamps with Gabriel server to synchronize time at the beginning of each experiment.

Figure 3.6 shows the time breakdown for all of the applications, in a normalized scale. Notice that there is a big variation in the total response time (number on top of each bar) across different applications. For most of them, 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. Moreover, except for Drawing As-

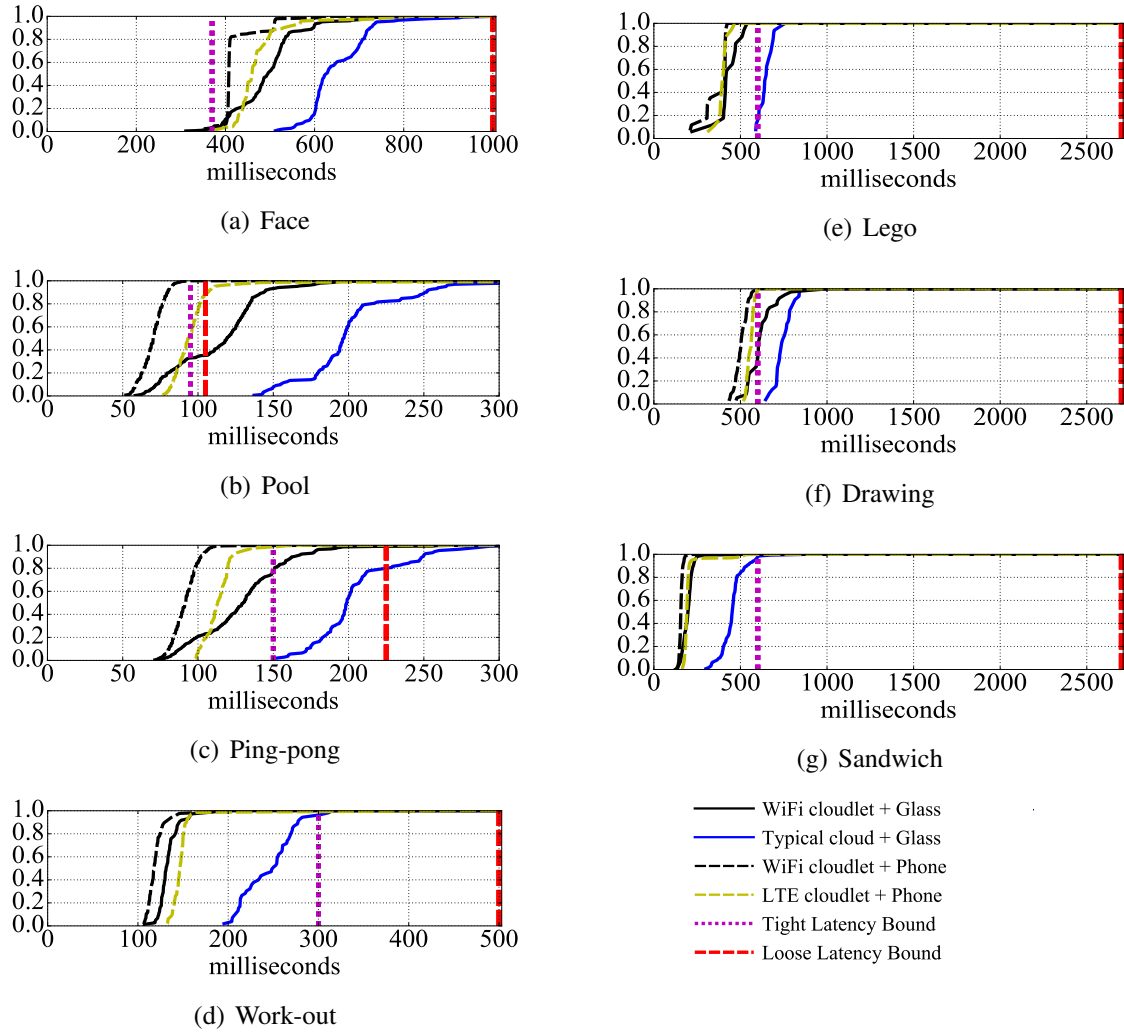


Figure 3.5: CDF of Application Response Time

sistant, guidance generation takes negligible time. Although this nontrivial step needs advanced task-specific knowledge, once the symbolic representation is available, guidance generation is computationally cheap. For example, for Pool Assistant, 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.

### 3.3.3 Benefits of Cloudlets

Is offloading to a cloudlet necessary to achieve such low response time, or does cloud offloading suffice? I run the application servers on three different Amazon EC2 sites- East (Virginia), West (Oregon), and Europe (Ireland). Note that our campus lab has unusually good connectivity to EC2-East (8 ms latency, 200 Mbps BW), so this should not be considered typical. Results for EC2-West can be considered typical for cloud offloading, as our connectivity to Amazon EC2 West is similar to the reported average RTT to optimal Amazon EC2 sites, 74ms [33]. The *solid*

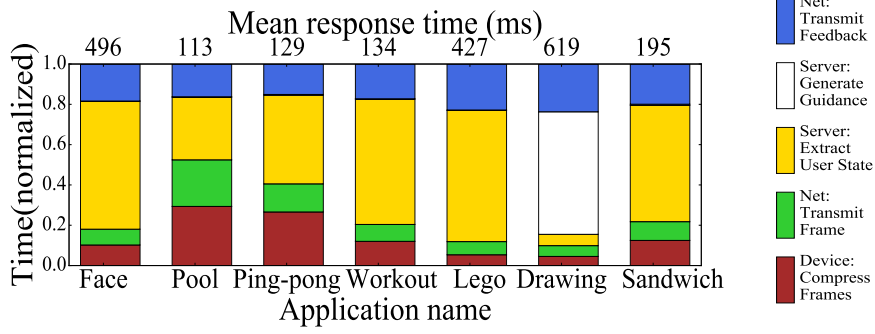


Figure 3.6: Breakdown of Application Response Time

*blue line* in Figure 3.5 corresponds to response time when offloading to EC2-West. In this case, only five of the seven applications meet their loose latency bounds, and among them only Workout and Sandwich Assistants meet their tight latency bounds. Face, Lego, and Drawing Assistant meet their loose latency bounds with the cloud, but are not even close to meeting their tight latency bounds. So in order to meet the tight latency bounds, or even come close for the Face application, it is necessary to use cloudlets.

Cloudlets were first explored to be placed close to a WiFi access point. Now, since deploying cloudlets close to cell towers is getting more traction [9], I also did experiments to see if such cloudlets placed in a cellular network would offer the same benefit as a WiFi cloudlet.

In order to investigate this, a 4G LTE network is set up in our lab for experimental use, with the permission from the FCC. The network uses a Nokia eNodeB to communicate with mobile devices. Since the Google Glass has no cellular radio hardware, I did experiments using the a Nexus 6 phone. The cloudlet machine connects to the eNodeB through a Nokia RACS gateway and local Ethernet.

The *dashed yellow line* in Figure 3.5 shows the CDF of system response time using the LTE cloudlet and the phone. All of the applications except Pool Assistant consistently meet their loose latency bounds. Five of them also meet their tight latency bounds. Although these results are better than the case of using Glass with cloudlet offload over WiFi, the improvement is due to the different devices being used. An apples-to-apples comparison with the case using the same phone to offload to cloudlets over WiFi (dashed black lines) shows that 4G LTE is slightly less effective than Wi-Fi in meeting the latency bounds of our benchmark suite. However, it is still a viable wireless technology for many applications.

## 4 Performance Optimization with Coarse-grain Parallelism

The previous section has shown that some applications are not fast enough to meet the latency bounds, and the server computation time is the main bottleneck. In this section, I study whether adding extra computing resources (e.g. CPU cores) could help reduce system latency. This is not an easy question as it might sound, since extra computing processes enabled by more hardware could usually only help to increase system throughput, but not reduce latency, especially if the algorithms are not designed to leverage multi-cores and distributed computing resources. This section introduces a novel approach to reduce system latency through combining different algorithms for the same cognitive processing task.

The approach is built on top of the coarse-grain parallelism structure in Gabriel. Multiple algorithms for the same task will run in parallel in different Cognitive VMs, each of which has different processing speed and accuracy level. A smart filter that runs in User Guidance VM will combine their results and generate final results with high speed and accuracy. This coarse-grain parallelism based approach doesn't require any modification to existing algorithm. As such, it could also offer speedup to proprietary software where source code is not available.

Admittedly, the new combined algorithm is not always better than each single algorithm being combined. It may be less accurate than some of them, and slower than some others. However, it offers a new tradeoff within the speed-accuracy space. And as will be shown in Section 4.2, this can be a good tradeoff – it can be almost as accurate as the most accurate algorithm and almost as fast as the fastest. Depending on the speed and accuracy requirements for a specific application, this new tradeoff point could be the ideal one.

In a real-world deployment scenario, a cloudlet will serve more than one user. In cases when a cloudlet is under-provisioned and spare resources are available, the approach suggested in the section can help to leverage the idle hardware to offer better user experience. However, when a cloudlet is over-provisioned and different users are competing for resources, this approach is no longer useful. The work in this section has not focused on minimizing the computing resources being used, and a careful management of cloudlet resources is beyond the scope of this thesis.

The remainder of this section introduces the approach for system performance optimization in detail. Although the focus is on reducing system latency (Section 4.2), an easier problem to increase system throughput is also discussed in Section 4.1, and the relationship between latency and throughput in the context of wearable cognitive assistance applications is shown.

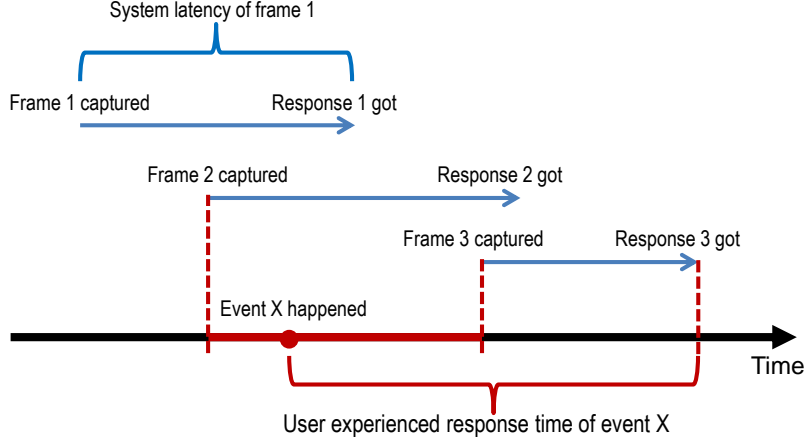


Figure 4.1: Difference Between System Latency and User Experienced Response Time (UERT)

## 4.1 Improving System Throughput

### 4.1.1 Throughput vs. User Experienced Response Time

Since user experience is only affected by the waiting time (i.e. latency) of a guidance, why do we care about system throughput? To answer this, we need to review the definition of system latency, or system response time.

The system response time in our 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, the response time a user has to experience can be longer. This is because with limited throughput, 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 4.1 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 include the system latency of one frame, and the average waiting time. Therefore, we define the User Experienced Response Time (UERT) as

$$UERT = \frac{1}{n} \sum_{i=1}^n (t_{end}^i - t_{start}^i) + \frac{1}{2} (t_{start}^i - t_{start}^{i-1}) \quad (4.1)$$

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

App	Latency (ms)	UERT (ms)
Face	498	735
Pool	114	176
Work-out	135	203
Ping-pong	130	191
Lego	428	631
Draw	622	679
Sandwich	215	322

Table 4.1: Comparison of System Latency and User Experienced Response Time

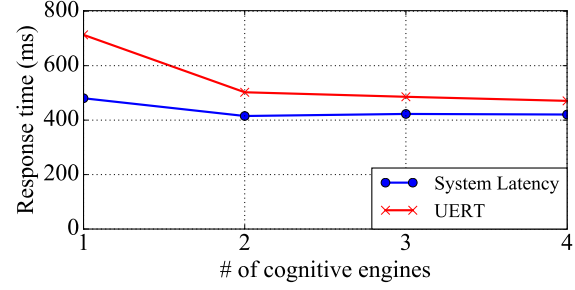


Figure 4.2: Improvement of UERT with Different Number of Engines

As can be seen from this definition, higher throughput will translate to lower UERT because it helps to reduce the interval the system picks up frames. Of course, this is assuming the average system latency to process one frame does not change because of the throughput change, which is usually true for a coarse-grain parallelism approach.

### 4.1.2 Experiment

First, I measure UERT of all the assistance applications according to equation 4.1. Table 4.1 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 use Face Assistant as an example application to demonstrate how to use the coarse-grain parallelism structure in Gabriel to increase system throughput and reduce UERT. To do this, I attached multiple Face applications as cognitive engines into Gabriel’s publish-subscribe backbone. The Control VM is modified so that it only feeds each input frame to one of the idle cognitive engines. As we can see in Figure 4.2, 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.

## 4.2 Improving System Latency

### 4.2.1 Background

The approach introduced in the previous section could only reduce UERT to as low as the system latency, but not any lower. For some of the applications, further speed optimization is needed, which requires shortening the system latency.

It is usually hard to leverage coarse-grain parallelism to improve system latency. Running multiple instances of the same algorithm concurrently only helps with throughput, but not latency. While rewriting existing code to exploit internal parallel structure of an algorithm would usually result in big improvements in computation speed, it usually takes a lot of extra work and makes it hard to keep in sync with the state-of-the-art algorithm in research community. In this section, we explore how to improve system latency solely based on coarse-grain parallelism, but

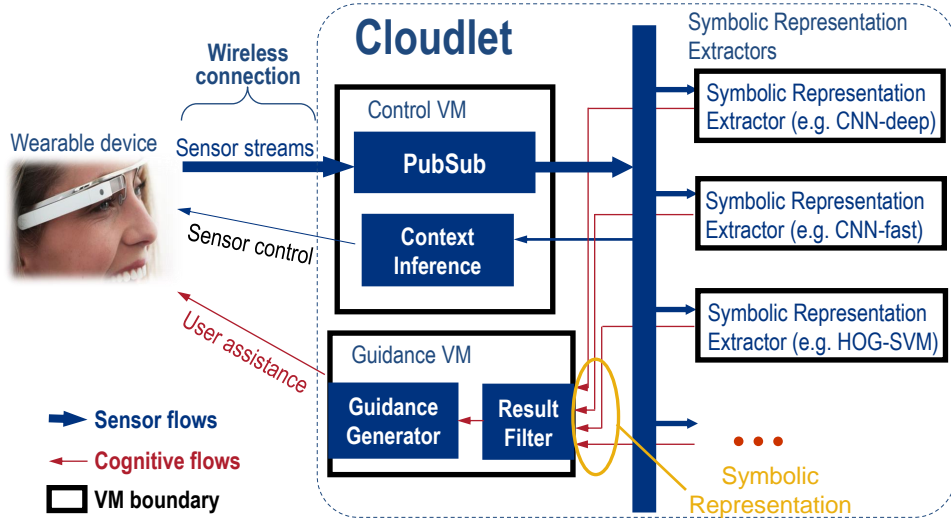


Figure 4.3: Gabriel Architecture for Combining Multiple Algorithm to Reduce Latency

with the help of multiple algorithms.

For any popular computer vision or signal processing task today, there usually exists more than one algorithm. A natural accuracy-speed tradeoff lies among them. For example, in an object classification task, a HOG+SVM approach is probably the simplest and fastest, but only gives modest accuracy. In the meantime, the most recent deep neural network (DNN) based approach could give much higher accuracy, but also takes longer time. Selecting an algorithm to use in practice means to give up accuracy or speed, leading to unsatisfactory user experience in either case.

Several existing works try to leverage multiple algorithms to achieve a better balance between speed and accuracy [10, 18, 55, 63, 64]. The basic idea is to run multiple algorithms, and to use the result from the faster (but less accurate) algorithm if the confidence level is very high (e.g. higher than 90%). Otherwise, the system will wait until the slower (but more accurate) algorithm to finish. This line of thinking fits perfectly with the coarse-grain parallelized architecture, since the different algorithms can run concurrently in different Cognitive VMs. However, it requires every algorithm, except the slowest one, to return a confidence level in addition to the processing result, which is not possible for all algorithms. Moreover, this approach treats each frame in a video independently and fails to leverage the correlation between contiguous frames. My goal, therefore, is to combine multiple algorithms that leverage correlation between frames, without the need of any confidence scores.

## 4.2.2 Approach

In this section, I propose a novel approach to combine different algorithms through coarse-grain parallelism. The core idea is to run multiple algorithms concurrently, and use 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. If not, we will wait until the slower one to finish.

The feasibility of this approach comes from temporal locality of a video stream: The environmental lighting, exposure level, background and objects in a scene, and camera viewpoint are not going to change very fast in a video capturing a user’s daily life. As some of these are the predominant factors affecting the accuracy of a computer vision algorithm, the accuracy of different algorithms would remain stable, too. For example, an object detection algorithm that does well in the past 3 seconds would probably suggest it will give accurate result in the next second.

Figure 4.3 shows how this approach can be implemented using 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` will decide which result can be trusted. If a result is deemed accurate, it will be sent to the `guidance generator` for task related guidance generation. Otherwise, the result is simply ignored.

Algorithm 1 describes the algorithm used in the result filter. Two important data structures are maintained: the *result\_history* is a dictionary keeping track of recent symbolic representation results from all algorithms, and the *confidence* is an array counting how many times each algorithm has been consecutively correct up till now. A main procedure, called `process`, is called every time the result filter receives a new result. It uses the `trust` function to decide whether the result can be used, based on the algorithm’s recent performance. If the result is from the best algorithm we have, it will be used to update confidence score of other algorithms through the `update_confidence` procedure.

The `trust` function and the `update_confidence` procedure can be implemented in different ways depending on different strategies to integrate results from multiple algorithms. The pseudocode presented in Algorithm 1 shows a simple version. The `trust` function simply compares the confidence score of an algorithm with a pre-defined threshold. If, and only if, an algorithm has been correct for the past *THRESHOLD* times, the next result it produces will be used. Note that in the real-time system, we don’t have a ground truth result to compare with. Therefore, each result is compared with the result from the best algorithm to determine if it is correct. The `update_confidence` procedure updates the confidence score. Only if the result from an algorithm matches that from the best algorithm on the same frame, will the confidence of that algorithm be increased. Otherwise, the confidence will be reset to zero. Since the results from different algorithms for the same frame are received at different time, the *result\_history* is used to preserve a short time history of results.

### 4.2.3 Early Results

The above mentioned approach can be applied to any cognitive assistance applications, as long as there exists multiple symbolic representation extraction algorithms. Before making the effort to implement the algorithms for the benchmark of cognitive assistance applications, I first created a simple application of image classification to study the feasibility of this approach. Thorough validation on the benchmark suite will be part of my future work to complete this thesis.

In the test application, the symbolic representation does simple image classification, which is a basic computer vision task and can be a component in a complex real-world application. Among hundreds of existing algorithms for this job, I picked two for my experiment. One



---

**Algorithm 1** Combining Algorithms

---

```
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]  $\geq$  THRESHOLD 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
```

---

Algorithms	Accuracy (%)	Average Latency (ms)
DNN-deep (A1)	91.8	621.4
DNN-classic (A2)	78.8	185.3
HOG-SVM (A3)	47.6	56.2
A1 + A2	90.0	313.3
A1 + A2 + A3	88.3	135.3

Table 4.2: Accuracy and Latency for Different Combination of Algorithms

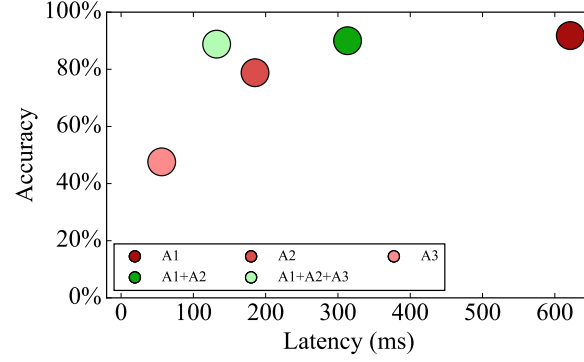


Figure 4.4: Accuracy and Latency Tradeoff for Different Algorithms

extracts HOG and color features and uses SVM for classification. The other uses deep neural network (DNN), which gives the state-of-the-art accuracy. For the latter approach, I also use two different network architectures with different depth, which results in different speed and accuracy properties. In summary, three different approaches are used: DNN-deep (A1), DNN-classic (A2), HOG-SVM (A3).

As we can see from Table 4.2, the DNN-deep approach, when running alone, can achieve the best accuracy of 91.8%, but is also the slowest approach. HOG-SVM, on the other hand, is extremely fast, resulting in system latencies less than 60ms. However, it only gives the correct classification result less than half of the time. DNN-classic is the one that sits in between, achieving reasonable accuracy (78.8%) and end-to-end latency (185.3ms). If we combine the two DNN based approaches and use DNN-deep to verify the dynamic accuracy of DNN-classic, the latency becomes 70% smaller than running DNN-deep alone, but the accuracy is only dropped by 1.8%. Similarly, if all three algorithms are run concurrently, the accuracy and latency become 88.3% and 135.3ms, respectively.

Figure 4.4 depicts the tradeoff between speed and accuracy for all the combination of algorithms. Each point in this two dimensional space represents an algorithm with a particular tradeoff. Ideally, we want a point that lies at the top-left corner of this graph, representing both high accuracy (top) and low latency (left). In this figure, we can clearly see that although A3 (HOG-SVM) is the fastest, it's far less accurate than others. On the other hand, A1 (DNN-deep) offers the highest accuracy, but it is the slowest. Combining three algorithms (A1+A2+A3) seems to provide a good tradeoff between speed and accuracy (closest to the top-left corner).

It is worth noting that there is no clear answer to which algorithm is the best. It all depends on the requirements of a specific application. For example, an algorithm that takes 10 times as much time and gains only 0.01% accuracy improvement can be the best if the cost of making an error is huge. Therefore, what I proposed here is not an approach to create the *best* algorithm, but rather an easy way to create a new point in the speed-accuracy tradeoff space that might be beneficial to many applications.

## 5 Path to Completion

There are three topics to be finished for the completion of the thesis. The first one is to build more wearable cognitive assistance applications to test the usability and generality of the Gabriel framework. I also need to extend the Gabriel framework to accommodate the new applications. The second topic is on optimization of system response time of Gabriel. The core idea and early experimental results are shown in Section 4.2, but a thorough validation remains. The last topic is to optimize Gabriel for reducing energy consumption of the mobile device. This is an crucial aspect to improve Gabriel’s usability.

### 5.1 Extend Gabriel with New Applications

So far, I have built seven prototype wearable cognitive assistance applications. These applications have not only tested the Gabriel framework’s system performance and confirmed the necessity of cloudlets, but also verified the ease of using Gabriel to prototype new applications. I am planning to build more applications of different characteristics to further diversify the applications in the benchmark suite.

The new applications will be carefully selected to have at least one of the following properties. First, the application will target addressing real-world problems. For example, I am planning to build an Ikea furniture assembly application that checks a user’s progress for every step. Second, the application will leverage sensors other than just cameras. A language translation application that recognizes a user’s speech and interpret it in real time is a good example. Third, the application will need to use more than one device in a collaboratively fashion. In the Work-out Assistant, for example, while the phone is used to capture a user’s action, his wrist watch can be used as a better device to offer guidance.

As new applications are developed, the Gabriel framework may need to be revised to better support these new applications. For example, if an application is dependent on multiple sensor streams, Gabriel needs to synchronize these data and offer application developers an easy-to-use interface to process multiple sensor streams at the same time. On top of this, the use of multiple devices will also complicate the system. Developers should have an easy approach to identify a device for input or output, and elegantly handle the situations when some devices are disconnected.

## 5.2 Improve System Latency

Section 4.2 has shown the basic idea of improving system latency by combining different algorithms through coarse-grain parallelism. The system will use the results from the faster but less accurate algorithm whenever possible, and use the results from the more accurate but slower algorithm if necessary. From a developer’s perspective, all of these algorithms are treated as black boxes. The only thing a developer has to do is to specify which algorithm should be used as dynamic ground truth to verify the accuracy of other algorithms. System speedup will then be achieved without any modification of original algorithm implementation.

Although I have tested the approach with an example application, thorough validation on a broader range of applications are needed. To do this, I plan to select a subset of my applications and design multiple symbolic representation extraction algorithms for each of them. For some applications where only one algorithm is available, I will test if simply using it with multiple parameter settings would still help.

Algorithm details to combine multiple algorithms also require further thought. For example, in the `trust` function, it may be possible to use some ambient sensor readings to determine our confidence about an algorithm result. It might also be beneficial to leverage perceptual hashing on input image to run a simple database lookup before running any symbolic representation extractor. I will run controlled experiments to test the effect of each of these parameters.

## 5.3 Optimization on Energy Consumption

### 5.3.1 Motivation

Reducing energy consumption of mobile applications has been a hot research topic for years. Running Gabriel client on a Google Glass results in running times of about 40 minutes of the device. This is obviously not long enough for people’s everyday usage. More importantly, when the mobile device is used with full load for some time, the heat will quickly accumulate on the device and may burn a user or cause discomfort.

LiKamWa et al. [34] have done a careful energy breakdown of Google Glass for multiple applications, in which they show that image sensing and WiFi transmission could consume as much energy as CPU processing. For devices with large screens (e.g. a smartphone or tablet), the energy consumed by screen can also be significant [51]. Therefore, besides reducing the CPU load of a mobile device through offloading, careful control of the sensing, network, and output components can also lead to significant energy savings.

In my current implementation of Gabriel client, the image sensor is always on to capture frames at the maximum framerate. The network also transmits as much data as the server could process. I believe a large portion of this work could be reduced to save energy. For example, for Ping-pong Assistant, there is no need to transmit frames at full framerate when the user is not in a rally. Moreover, the Lego Assistant could probably take a rest for a while after each guidance is given, because the user needs at least a few seconds to complete a step. Therefore, my goal is to create a dynamic, application-guided power control mechanism for the mobile device.

### 5.3.2 Approach

The simplest idea to save energy is to turn off the sensors and stop transmitting data when they are not needed by any application. Alternatively, it can also lower the sensing and transmission rate. However, if the sensor cannot be turned on again at the right time, or if the sensing rate is too low, the application might miss important events that have happened to a user and fails to provide prompt guidance.

It is usually easier for an application to decide when to turn off a sensor than to turn it on. In the example of a Lego Assistant, sensors could be turned off when a new guidance is given, but the time they should be back on depends on how fast the user could finish a step. It may be possible to estimate the interval based on other users' performance and adjust the estimation as the user uses the application. Another approach is to use the information from low energy-consuming sensors (e.g. accelerometers) to activate high energy-consuming sensors (e.g. cameras) with pre-defined events (e.g. user remains stable or starts moving).

In Gabriel, the *context inference* module has been designed to understand an application's requirements and control the sensors on a mobile device. Although the mechanism for sensor control has been implemented, currently there is no clear approach to explain policies by each application. I need to study the needs and properties of common applications to provide an easy-to-use API. Moreover, the implementation of some mechanisms may require additional processing from the client side, such as continuous monitoring of the state of a low energy-consuming sensor. This is in line with the thinking in Offload Shaping work [24].

## 5.4 Topics Not Covered

To help clarify the scope of this thesis, here is a list of topics that the thesis will Not cover.

- *The thesis does not provide mechanisms to simplify writing computer vision or signal processing code.* The Gabriel architecture only provides common system functionalities such as user communication, PubSub backbone for parallelism, and coordinating with multiple devices. The application-specific algorithms will need to be designed manually.
- *Gabriel does not support multi-user collaborative applications.* Although some applications described in the thesis will involve the use of multiple devices (e.g. phones, watches, or glasses), they all serve one user. Those cognitive assistance tasks that need information exchange between users will not be included.
- *Gabriel cannot support applications with millisecond level response time requirement.* Gabriel aims at supporting low latency applications, but there is a limit on how low the latency can be. As shown in Section 2.2, Gabriel will have around 30ms latency given zero application-specific processing time. Although this number will get smaller as client hardware and network technology get better, the total system latency is not likely to get below 10 millisecond.
- *The applications in the benchmark suites are instance based and do not generalize well for different object types.* For example, the detection of objects such as bread or hamburger in Sandwich Assistant can only work with the toy sandwiches we have, and will not au-

tomatically work with real food. Building smarter detectors that understand the semantic meaning of "bread" is beyond the scope of the thesis.

- *This thesis does not discuss constraints and optimizations for efficiently using cloudlet resources.* Although a cloudlet has much more computation resources than a mobile device, it is not infinite, especially when shared by multiple users. This thesis focuses on optimizing system performance for one user assuming abundant resources in the cloudlet, and will not address cases when a cloudlet is over provisioned.

## 5.5 Timeline

Timeline		Plan
2016 Summer	July – Aug.	Optimize Gabriel’s system latency
2016 Fall	Sept. – Nov. Dec.	Implement new Gabriel applications MobiSys submission
2017 Spring	Jan. – May Apr.	Optimize Gabriel’s energy consumption SenSys submission
2017 Fall	Aug. – Nov. Nov. Dec.	Thesis writing Finish thesis dissertation Thesis Defense

Table 5.1: Timeline for Thesis Completion

## 6 Related Work

### 6.1 Wearable Cognitive Assistance

There have been many efforts to offer cognitive assistance through a wearable device. For example, Chroma [58] uses Google Glass to provide color adjustments for the color blind. Cross-Navi [50] uses smart phone to guide blind people to cross the road. Siewiorek et al. [53] describes Virtual Coaches that provide proactive assistance to handicapped people using sensor data (e.g. pressure sensor). Opportunity Knocks [42] offers navigation assistance to help those with mild cognitive disabilities. Finally, SenseCam [22] introduces a wearable image-based life logging system for memory rehabilitation.

Most of these applications run completely on the mobile device. As a result, their functionalities are constrained by the limited computing power and battery capacity of the device. Recently, applications like Siri have sought to offload complex computation to the cloud, which then suffer from the long latency incurred by network transmission. This thesis targets at enabling a new genre of wearable applications that require both high computation and crisp system response. This requires careful design of mobile-cloud system architecture and optimization on both network transmission and algorithm computation.

### 6.2 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 [37] 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. [61] and Lai et al. [31] reported on the impact of latency on thin-client user experience. In 2015, Taylor et al. [60] 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. [11] studied how the use of cloudlet 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. [23] and Nah [38].

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.

## 6.3 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. Flinn [14] 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, Amazon Silk, and Apple Siri. Among recent research efforts, MAUI [12] dynamically selects an optimal partitioning of a C# program to reduce the energy consumption of a mobile application. COMET [16] uses distributed shared memory to enable transparent offloading of Android applications. Odessa [44] 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 aims to support the widest possible range of applications without restrictions on their programming languages or operating systems.

## 6.4 Speeding up Computer Vision

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 GPUs. However, not all algorithms could benefit from this approach, since the improvement is limited by the degree of internal parallelism. In addition, such approaches usually require a complete rewrite of existing algorithm code. Several approaches have been proposed to simplify the process to leverage parallelism. For example, Halide [45] provides a cross-platform language and compiler for easy experimentation of image processing workflows. However, these work usually incurs restrictions on how an algorithm is structured.

Another thread of work has been trying to leverage a coarser granularity of parallelism. They run multiple algorithms 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 [64], a substantial body of work has been using cascade approach to speed up image classification, object detection, and OCR tasks [10, 18, 55, 63]. 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 requires every algorithm to have a con-



confidence score to filter out samples, which is not possible for all algorithms. Moreover, these algorithms treat each input image independently, thus miss the opportunity to take advantage of any hints from earlier images. In contrast, this thesis proposes a new optimization strategy that does not require any confidence score, and takes advantage of the temporal coherence between nearby frames.

# Bibliography

- [1] Fractional-ball aiming. <http://billiards.colostate.edu/threads/aiming.html#fractional>. Accessed on November 27, 2015.
- [2] Issue 512: Bluetooth connection interfering with TCP Traffic on WiFi. <https://code.google.com/p/google-glass-api/issues/detail?id=512>, May 2014. 3.3.1
- [3] 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. 1.1
- [4] Brandon Amos, Jan Harkes, Padmanabhan Pillai, Khalid Elgazzar, and Mahadev Satyanarayanan. OpenFace: Face Recognition with Deep Neural Networks. <http://github.com/cmusatyalab/openface>, 2015. Accessed: 2015-11-11.
- [5] 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.
- [6] 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. 1.3
- [7] 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. 2.1.2
- [8] 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. 3.2, 3.2.2
- [9] Gabriel Brown. Converging Telecom & IT in the LTE RAN. White Paper, Heavy Reading, February 2013. 1, 2.1.2, 3.3.3
- [10] 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. 4.2.1, 6.4
- [11] Sarah Clinch, Jan Harkes, Adrian Friday, Nigel Davies, and Mahadev Satyanarayanan. How close is close enough? understanding the role of cloudlets in supporting display appropria-

- tion by mobile users. In *Pervasive Computing and Communications (PerCom), 2012 IEEE International Conference on*, pages 122–127. IEEE, 2012. 6.2
- [12] 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. 6.3
  - [13] 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
  - [14] Jason Flinn. Cyber foraging: Bridging mobile and cloud computing. *Synthesis Lectures on Mobile and Pervasive Computing*, 7(2):1–103, 2012. 1.1, 1.2, 6.3
  - [15] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.
  - [16] 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. 6.3
  - [17] 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. 2.1.2
  - [18] Alexander Grubb and Drew Bagnell. Speedboost: Anytime prediction with uniform near-optimality. In *AISTATS*, volume 15, pages 458–466, 2012. 4.2.1, 6.4
  - [19] 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. 2.1.2
  - [20] Kiryong Ha, Zhuo Chen, Wenlu Hu, Wolfgang Richter, Padmanabhan Pillai, and Mahadev Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 68–81. ACM, 2014. 3.3.1
  - [21] Cristy Ho and Charles Spence. Verbal interface design: Do verbal directional cues automatically orient visual spatial attention? *Computers in Human Behavior*, 22(4):733–748, 2006. 3.2.2
  - [22] Steve Hodges, Emma Berry, and Ken Wood. SenseCam: A wearable camera that stimulates and rehabilitates autobiographical memory. *Memory*, 19(7):685–696, 2011. 6.1
  - [23] 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. 6.2
  - [24] Wenlu Hu, Brandon Amos, Zhuo Chen, Kiryong Ha, Wolfgang Richter, Padmanabhan Pil-

- lai, 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. 2.1.4, 5.3.2
- [25] 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.
- [26] 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.
- [27] 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. 3.2.1
- [28] 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.
- [29] Davis E King. Dlib-ml: A machine learning toolkit. *The Journal of Machine Learning Research*, 10:1755–1758, 2009.
- [30] 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. 3.2.2, 3.2.3
- [31] 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. 6.2
- [32] Michael B Lewis and Andrew J Edmonds. Face detection: Mapping human performance. *Perception*, 32(8):903–920, 2003. 1.1
- [33] 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. 2.1.2, 3.3.3
- [34] 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. 5.3.1
- [35] 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
- [36] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014. 2.1.4
- [37] 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. 3.2.1, 3.2.3, 6.2
- [38] 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. 6.2

- [39] OpenStack. <http://www.openstack.org/>, February 2015. 3.3.1
- [40] Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *Knowledge and Data Engineering, IEEE Transactions on*, 22(10):1345–1359, 2010.
- [41] 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. 3.2
- [42] 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. 6.1
- [43] Michael I Posner and Steven E Petersen. The attention system of the human brain. Technical report, DTIC Document, 1989. 1.3
- [44] 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. 6.3
- [45] 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. 6.4
- [46] Meike Ramon, Stephanie Caharel, and Bruno Rossion. The speed of recognition of personally familiar faces. *Perception*, 40(4):437–449, 2011. 1.1, 3.2.1
- [47] Gavriel Salvendy. *Handbook of human factors and ergonomics*. John Wiley & Sons, 2012. 3.2
- [48] 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. 2.1.2
- [49] 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. 3.2.2
- [50] 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. 6.1
- [51] 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. 5.3.1
- [52] Ben Shneiderman and Catherine Plaisant. *Designing the user interface (4th edition)*. Pearson Addison Wesley, USA, 1987. 3.3(a)
- [53] Daniel Siewiorek, Asim Smailagic, and Anind Dey. Architecture and applications of virtual

- coaches. *Quality of Life Technology Handbook*, page 197, 2012. 6.1
- [54] Ray Smith. An overview of the tesseract ocr engine. In *icdar*, pages 629–633. IEEE, 2007. 1.2, 1.3
  - [55] 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. 4.2.1, 6.4
  - [56] Heikki Summala. Brake reaction times and driver behavior analysis. *Transportation Human Factors*, 2(3):217–226, 2000. 3.2
  - [57] 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.
  - [58] 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, 6.1
  - [59] 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. 3.2
  - [60] Brandon Taylor, Yoshihisa Abe, Anind Dey, Mahadev Satyanarayanan, Dan Siewiorek, and Asim Smailagic. Virtual machines for remote computing: Measuring the user experience, 2015. 6.2
  - [61] Niraj Tolia, David G Andersen, and Mahadev Satyanarayanan. Quantifying interactive user experience on thin clients. *Computer*, 39(3):46–52, 2006. 6.2
  - [62] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of cognitive neuroscience*, 3(1):71–86, 1991. 1.3
  - [63] Nuno Vasconcelos and Mohammad J Saberian. Boosting classifier cascades. In *Advances in Neural Information Processing Systems*, pages 2047–2055, 2010. 4.2.1, 6.4
  - [64] 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. 4.2.1, 6.4