

An Empirical Study of Latency in an Emerging Class of Edge Computing Applications for Wearable Cognitive Assistance

Zhuo Chen, Wenlu Hu, Junjue Wang, Siyan Zhao,
Brandon Amos, Guanhang Wu, Kiryong Ha, Khalid Elgazzar,
Padmanabhan Pillai[†], Roberta Klatzky, Daniel Siewiorek, Mahadev Satyanarayanan
Carnegie Mellon University and [†]Intel Labs

ABSTRACT

An emerging class of interactive wearable cognitive assistance applications is poised to become one of the key demonstrators of edge computing infrastructure. In this paper, we design seven such applications and evaluate their performance in terms of latency across a range of edge computing configurations, mobile hardware, and wireless networks, including 4G LTE. We also devise a novel multi-algorithm approach that leverages temporal locality to reduce end-to-end latency by 60% to 70%, without sacrificing accuracy. Finally, we derive target latencies for our applications, and show that edge computing is crucial to meeting these targets.

CCS CONCEPTS

• **Human-centered computing** → **Empirical studies in ubiquitous and mobile computing**; *Ubiquitous and mobile computing systems and tools*; • **Software and its engineering** → **Distributed systems organizing principles**; • **Networks** → *Wireless access points, base stations and infrastructure*; *Mobile networks*; Network measurement; • **Computer systems organization** → *Real-time system architecture*;

KEYWORDS

Cloudlet, Edge Computing, Mobile Computing, Cloud Computing, Smart Glass, Augmented Reality, HoloLens

1 Introduction

One of the earliest motivations of edge computing was to lower the end-to-end latency of cloud offload. As early as 2009, it was recognized that deploying powerful cloud-like infrastructure just one wireless hop away from mobile devices could be transformative [30]. We use the term *cloudlet* to refer to an instance of such infrastructure. Many companies are

now exploring commercial deployments of cloudlets based on Wi-Fi as well as 4G LTE. The term *fog computing* has also been used for computing at the edge of the Internet, and *fog node* for the associated hardware infrastructure [5].

In this paper, we examine the end-to-end latency characteristics of an emerging class of resource-hungry applications whose processing demands could not be met without edge computing. In particular, their resource demands exceed those of mobile devices even projected well into the future. Off-loading computation is thus essential. At the same time, end-to-end latency matters because they are human-in-the-loop systems that deeply engage user attention. These *wearable cognitive assistance* applications stream sensor data from a wearable device to a cloudlet, perform real-time compute-intensive processing, and return just-in-time task-specific guidance to the user [10]. In effect, these applications bring AI technologies such as computer vision, speech recognition, natural language processing, and deep learning within the inner loop of human cognition and interaction. Since 2014, we have built nearly a dozen such applications, along with a common underlying platform that supports them.

Our goal in this paper is to obtain an empirical understanding of end-to-end latency in these applications. Where does the time go? How much would using 4G LTE affect results? How powerful do cloudlets have to be in order to support such applications? Are specialized hardware such as GPUs essential in a cloudlet? Where should we focus effort in order to improve the user-perceived performance of the system? How hard do we need to work in lowering end-to-end latency? Can we leverage additional processing resources to improve latency rather than just throughput? These are examples of the kinds of questions that we answer in this study. Since this is the very first empirical study of its kind, it can only provide us with an initial set of answers to such questions. However, even preliminary answers to important questions are valuable in a new and emerging area in order to provide guidance for system designers and software developers. As more applications emerge over time, built by many research groups and possibly commercial entities, a broader and more comprehensive empirical study can be conducted to revisit these questions. We therefore view this paper as providing the first insights, rather than the last word, in this important domain of the future.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SEC '17, October 12–14, 2017, San Jose / Silicon Valley, CA, USA

© 2017 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-5087-7/17/10.

<https://doi.org/10.1145/3132211.3134458>

App Name	Example Input Video Frame	Description	Symbolic Representation	Example Guidance
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 fractional aiming system [1]. Color, line, contour, and shape detection are used. The symbolic representation describes the positions of the balls, target pocket, and the top and bottom of 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 representation is a 3-tuple: in rally or not, opponent position, ball position. Whispers “left” or “right” or offers spatial audio guidance using [34]. Video URL: https://youtu.be/_lp32sowyUA	<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 [17] on a 10-15 frame video segment to classify the exercise. Uses smart phone on the floor for third-person viewpoint.	<Action, count>	Says “8 ”
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 state-of-art face recognizer using deep residual network [11]. Whispers the name of a person. Can be used in combination with Expression [2] to offer conversational hints.	ASCII text of name	Whispers “Barack Obama”
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. Color normalization is needed. The symbolic representation is a matrix representing color for each brick. Video URL: https://youtu.be/7L9U-n29abg	[[0, 2, 1, 1], [0, 2, 1, 6], [2, 2, 2, 2]]	 Says “Put a 1x3 green piece on top”
Draw		Helps a user to sketch better. Builds on third-party app [14] that was originally designed to take input sketches from pen-tablets and to output feedback 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. Displays the error alignment in sketch on Glass. Video URL: https://youtu.be/nuQpPtVJC6o		
Sandwich		Helps a cooking novice prepare sandwiches according to a recipe. Since real food is perishable, we use a food toy with plastic ingredients. Object detection follows the state-of-art faster-RCNN deep neural net approach [29]. Implementation is on top of Caffe [15] and Dlib [18]. Transfer learning [24] helped us save time in labeling data and in training. Video URL: https://youtu.be/USakPP45WvM	Object: “E.g. Lettuce on top of ham and bread”	 Says “Put a piece of bread on the lettuce”

Figure 1: Prototype Wearable Cognitive Assistance Applications Studied in This Paper

2 Application Attributes and Implementation

As wearable cognitive assistance is just an emerging concept, there are not many existing applications in this space, let alone open-source ones that cover a broad range of usage scenarios. As a result, we found it necessary to develop our own broad set of cognitive assistance applications covering a variety of assistive tasks. Out of the nearly dozen apps we have built, we picked a subset of seven that are representative, as summarized in Figure 1. YouTube video demos of some applications are available at <http://goo.gl/02m0nL>.

2.1 Application Diversity and Similarity

Our applications span a wide variety of guidance tasks. All use computer vision (CV), but with substantially different algorithms, ranging from color and edge detection, face detection and recognition, to object recognition based on deep neural networks (DNNs). Some, such as Lego and Sandwich, provide step-by-step guidance to completing a task. They determine whether the user has correctly completed the current step of the task, and then provide guidance for the next step, or indicate corrective actions to be taken. In this regard, they resemble GPS-navigation, where step-by-step instructions are provided, and corrective actions are suggested if the user makes a mistake or deviates from the plan.

Other applications are even more tightly interactive. For example, Ping-pong provides guidance on whether the user should hit left or right based on the opponent position and ball trajectory during a rally. Pool provides continuous guidance as the player aims the cue stick. For both, the feedback needs to be immediate, and thus requires very low end-to-end latency for the entire cognitive assistance processing stack.

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

Additionally, all of these applications utilize a similar 2-phase operational structure. In the first phase, the sensor inputs are analyzed to extract a *symbolic representation* of task progress (fourth column in Figure 1). This is an idealized representation of the input sensor values relative to the task, and excludes all irrelevant detail. This phase has to be tolerant of considerable real-world variability, including variable lighting levels, changing camera positions, task-unrelated background clutter, 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 task-specific state space. In the second phase, the symbolic representation is further analyzed by custom application logic to generate appropriate guidance.

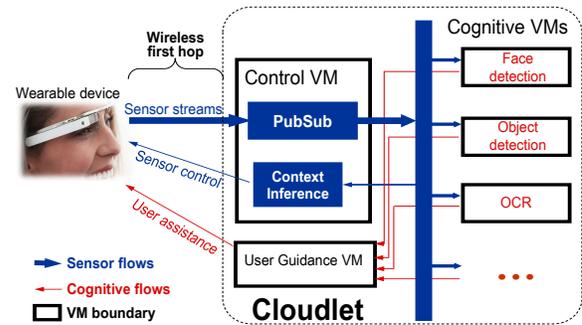


Figure 2: Gabriel Architecture

2.2 Gabriel Platform

Due to the structural similarities mentioned above, we are able to implement all seven applications on top of *Gabriel* [10], an open-source offloading framework that can be run in the cloud or on a cloudlet (Figure 2). In this architecture, a mobile device does basic preprocessing of sensor data (e.g. compression and encoding), and then streams this over a wireless network to the back-end which is organized as a collection of virtual machines (VMs). A single *control VM* is responsible for all interactions with the mobile device. Multiple VM-encapsulated *cognitive engines* concurrently process the incoming sensor data. The application-specific computer vision algorithms (i.e., the phase 1 processing) in our applications are implemented within cognitive VMs. Gabriel uses a publish-subscribe (PubSub) mechanism to decode and distribute the sensor streams to the cognitive VMs. A single *User Guidance VM* integrates cognitive VM outputs and performs higher-level cognitive processing (i.e., the phase 2 processing steps). From time to time, this triggers output for user assistance. We have experimentally confirmed that the VM-based Gabriel framework adds negligible delay to the response times of interactive applications. Lighter weight encapsulation such as Docker containers is also possible in our implementation, but does not change our results significantly.

The Gabriel front-end, which runs on the mobile device, captures 640x360 image frames at 15fps. Depending on the network conditions and the processing speed of the cognitive VMs, only a subset of the frames are streamed to the cloud or cloudlet in MJPEG format. The front-end is also responsible for presenting guidance to the user. The guidance messages are in JSON format that usually consist of a few hundreds of bytes. These are images that are displayed or verbal cues generated using text-to-speech on the device. Since all of the application-specific components lie within the Gabriel back-end, it is straightforward to run Gabriel applications on a diverse set of devices by simply porting the Gabriel front-end. We have successfully run our applications using Android phones, Google Glass, Microsoft HoloLens, Vuzix Glass, and ODG R7 device.

	CPU	RAM
Cloudlet	Intel® Core™ i7-3770 3.4GHz, 4 cores, 8 threads	15GB
Cloud	Intel® Xeon® E5-2680v2 2.8GHz, 8 VCPUs	15GB
Phone	Krait 450, 2.7 GHz, 4 cores	3GB
Google Glass	TI OMAP4430, 1.2 GHz, 2 cores	2GB
Microsoft HoloLens	Intel® Atom™ x5-Z8100 1.04 GHz, 4 cores	2GB
Vuzix M100	TI OMAP4460, 1.2 GHz, 2 cores	1GB
ODG R7	Krait 450, 2.7 GHz, 4 cores	3GB

Figure 3: Hardware Used in Experiments

In summary, our suite of applications covers a broad, representative range of the emerging class of interactive cognitive assistance applications. As our applications support multiple client devices, and both cloud and cloudlet back-ends, they can help us examine how various aspects of the system affect performance. In particular, we answer the following questions:

- How much does edge computing affect end-to-end latencies of these applications?
- How does edge computing based on cellular/LTE compare with that based on WiFi?
- Does the choice of end-user device affect performance?
- How much can hardware accelerators and extra CPU cores in the back-end help?
- Short of devising revolutionary new algorithms, what can we do to improve cloud/cloudlet processing time?
- How hard do we need to work at reducing latency?

In the rest of this paper, we obtain answers to these questions in the context of our applications.

3 Latency Measurements

We evaluate the performance of our suite of wearable cognitive assistance applications in terms of end-to-end latency, including processing and network time, for different configurations of the system. We compare offloading the application back-ends to different sites, including the public cloud, a local cloudlet over WiFi, and a cloudlet on a cellular LTE network. We also investigate how mobile hardware affects networking and client computation time. By timestamping critical points of an application, we obtain a detailed time breakdown for it, revealing system bottlenecks and optimization opportunities.

3.1 Experimental Setup

The specifications of the hardware used in our experiments are shown in Figure 3. We have set up a desktop-class machine

running OpenStack [23] to represent a cloudlet. For applications (e.g. Sandwich) that use GPU, we use an NVIDIA GeForce GTX 1060 GPU (6GB RAM).

To study the relative benefits of edge computing compared to using a centralized cloud, we use C3.2xlarge VM instances in Amazon EC2 as cloud servers. These are the fastest available in terms of CPU clock speed in January, 2017.

We use Nexus 6 phones as stand-ins for high-end wearable hardware in most of our experiments. We also experiment with Google Glass, Microsoft HoloLens, Vuzix M100 Glass, and ODG R7 Glass to demonstrate how client hardware affects latency. They connect to the server over a dedicated WiFi access point using 802.11n with 5GHz support whenever possible. If a device (such as Google Glass) does not support 802.11n WiFi over 5GHz, we use the best WiFi supported by it (e.g. 802.11b/g on 2.4 GHz for Google Glass). All of the devices except HoloLens run a variant of Android, while HoloLens runs Windows 10. The Gabriel front end has been implemented on both OSes, and performs identical processing steps on all devices.

Each experiment typically runs for five minutes. For reproducibility, the mobile device sends pre-captured frames instead of live frames, but keeps its camera on. For consistent results, we use ice packs to cool the mobile device to avoid CPU clock rate fluctuation due to thermal throttling [10].

3.2 Baseline Measurements

We first evaluate how well our applications perform in a baseline edge-computing configuration. Here, the back-end services are run on a WiFi-connected cloudlet machine, and we use a phone as the client device. This serves as our control scenario to which we will compare other system setups that vary particular system parameters.

Figure 4 shows the cumulative distribution functions (CDFs) of response times of each of our applications under various system setups. The total end-to-end latency is measured on the wearable device from when the frame is captured to when the corresponding guidance / response is received. This does not include the time to present the guidance (e.g. text-to-speech, image display) to the user. The solid black lines correspond to our baseline settings. These curves are almost always the leftmost in each plot, indicating this configuration results in the best latencies among the different system settings. We will investigate other configurations later in this section.

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

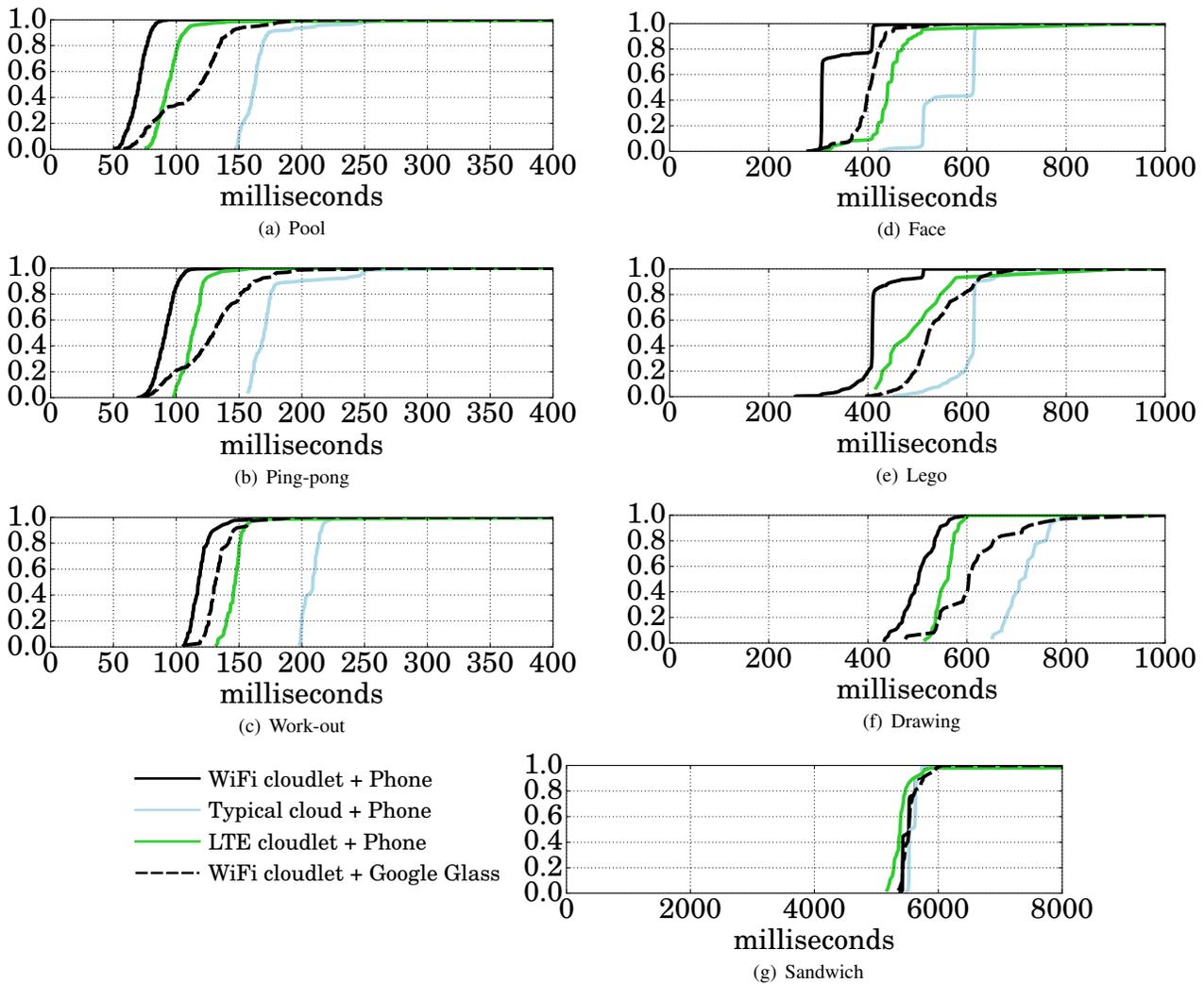


Figure 4: CDF of Application End-to-end Latency

3.3 Cloudlet vs. Cloud

We next ask the question, how are latencies affected if edge computing infrastructure is not available, and the application back-ends are forced to run in a distant cloud data center? We repeat our experiments, using an Amazon AWS server to host the application back-ends. We use an instance in AWS-West to represent a typical public cloud server, as the round trip time (RTT) from our test site to AWS-West correspond well to reported global means of 74 ms [20].

The solid blue lines in Figure 4 indicate latencies when offloading to the cloud. In general, offloading to cloudlets is clearly a win for our benchmark suite. Looking more closely, cloud offload almost always incurs an additional 100 to 200 ms latency compared to using a cloudlet. For Pool, the 90th percentile latency of cloud offloading is more than 2x that of cloudlet offloading, and will likely make a big

difference in user experience. On the other hand, the difference in offloading sites makes negligible impact to Sandwich, as the total compute time dwarfs these differences.

How much of the performance difference is due to the network versus other differences between the cloud and cloudlet? To investigate this, we update the processing pipeline to log timestamps, and provide a breakdown of where time is spent in each application (shown in Figure 5). Starting from bottom to top, the breakdown components are: (1) compressing a captured frame on the mobile device; (2) transmitting the image to the server; (3) extracting the symbolic representation on the server; (4) generating guidance on the server; and (5) transmitting guidance back to the mobile device. The bar height (number on top of each bar) represents the average end-to-end latency of the application. In order to correctly measure the network transmission time, we synchronize time between the

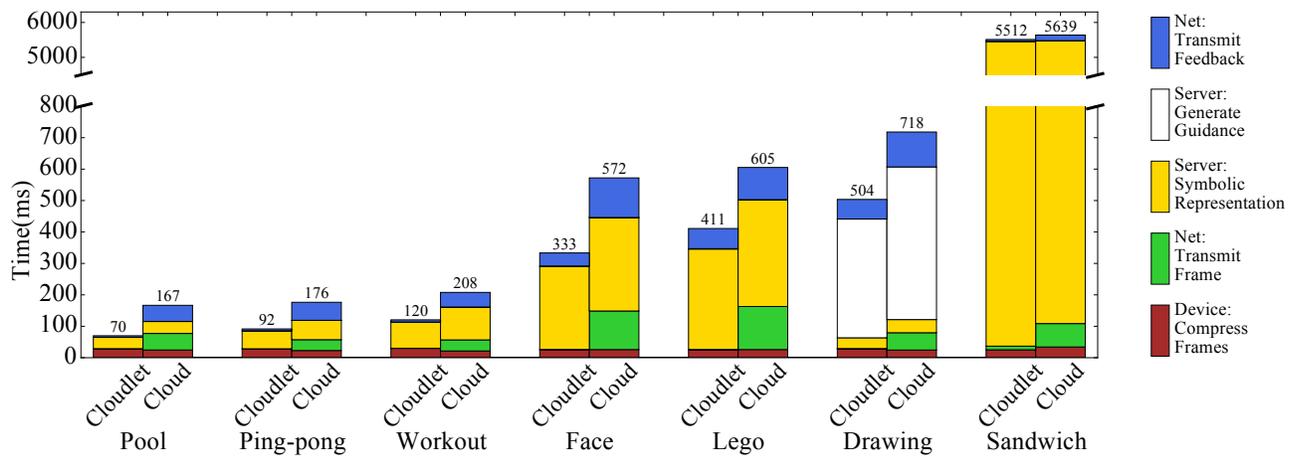


Figure 5: Mean Latency Breakdown - Cloudlet vs. Cloud for Phone over WiFi

Gabriel server and the client by exchanging timestamps at the beginning of each experiment.

The time breakdowns in Figure 5 show some interesting insights. First of all, for the baseline cloudlet case (left bars for each application), relatively little time is spent on network transmission. This is the advantage that edge computing promises – cloud-like resources with low latency and high bandwidth, resulting in low network overheads. In contrast, with cloud, the network overheads are high, and a significant fraction of the time is spent on transmission for most applications. Up to 40% more time is spent on network transmissions in the cloud case. Note that all of the client devices employ aggressive power management of the radio link. The networking times include overheads of transitioning out of low-power states, so some of the numbers (e.g. transmit feedback) may be larger than expected.

Secondly, the time spent on server compute time remains almost unchanged between cloudlet and cloud cases. For most applications, almost all compute time is devoted to extracting the symbolic representation, while generation of user guidance is computationally trivial, and is not visible in the plots. The one exception is Draw, which has relatively modest computational requirements for extracting state, but runs a complex, third-party application to generate guidance.

Finally, Sandwich remains the big outlier, with far higher response times than the other applications. Its time breakdown is dominated by the symbolic representation extraction step, which averages over five seconds for both the cloud and cloudlet implementations, due to the use of a computationally expensive deep neural network. All other components are relatively insignificant, although close inspection of the plots do show that network transmission increases in the cloud case.

Overall, our measurements show a very distinct, end-to-end response time advantage when using edge compute resources rather than the public cloud across our suite of applications.

3.4 4G LTE vs. WiFi for First Hop

The initial research efforts on edge computing focused on cloudlets connected to local WiFi networks. This connectivity provides excellent bandwidth and low latency to mobile devices on the local WiFi network. However, this is not the only way to deploy edge infrastructure. In particular, as part of a larger push toward network function virtualization (NFV) and software defined networking (SDN), the telecommunications industry is pursuing plans to install general-purpose compute infrastructure within cellular networks [8], enabling LTE-connected edge computing services. How much does LTE as the first wireless hop affect application latency?

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

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

The solid green lines in Figure 4 show the CDFs of system response times using the LTE-connected cloudlet configuration. These lines are consistently to the right of the lines corresponding to WiFi-connected cloudlets, with about 30

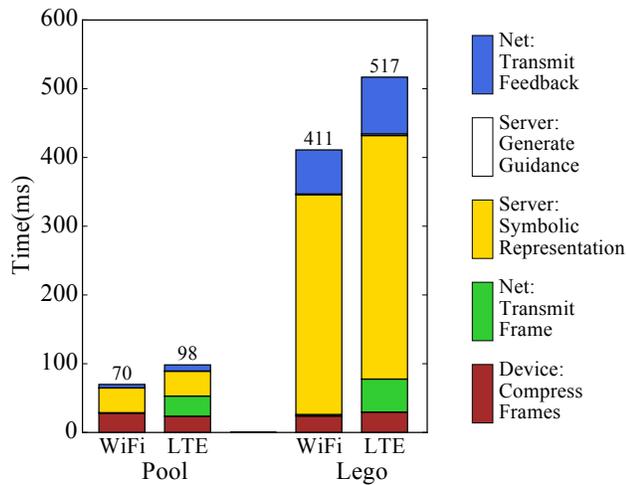


Figure 6: Latency Breakdown - WiFi vs. LTE Cloudlets

milliseconds difference. Excluding Sandwich, the relative difference of the 90th percentile latency ranges from 6.8% (Draw) to 33.8% (Pool). These differences are consistent with expectations, as 4G LTE has longer network latency (as measured by ping), and lower bandwidth compared to WiFi.

We also plot the time breakdown comparing WiFi and LTE scenarios in Figure 6. Due to space limitations, we only show results for two applications: Pool, which is the most interactive one, and Lego, which is one of the slower ones. The figure confirms that most of time difference is due to increase in network transmission times in LTE.

Overall, although LTE cloudlets are at a disadvantage when compared to WiFi cloudlets, they do provide reasonable application performance. In particular, Figure 4 shows that the latencies are better with LTE cloudlets than with cloud-based execution for all of the applications studied. Thus, we believe that LTE cloudlets are a viable edge computing strategy and can provide significant benefits over the default of cloud off-loading. Furthermore, if 5G lives up to expectations of greatly reduced radio network latencies, and the promises of greater bandwidth, the performance with cloudlets in the cellular network will improve even more.

3.5 Mobile Hardware

Since most processing is offloaded to the Gabriel back-end, one expects the choice of mobile device to have little impact on end-to-end latency. To confirm this intuition, we repeat the latency measurements of our applications running on a WiFi cloudlet, but replace the phone with Google Glass as the client device. The results are shown by the dashed black lines in Figure 4. Surprisingly, using Glass has a profound effect on the latencies, increasing them across the board for all applications. For Pool, the 90th percentile latency increases

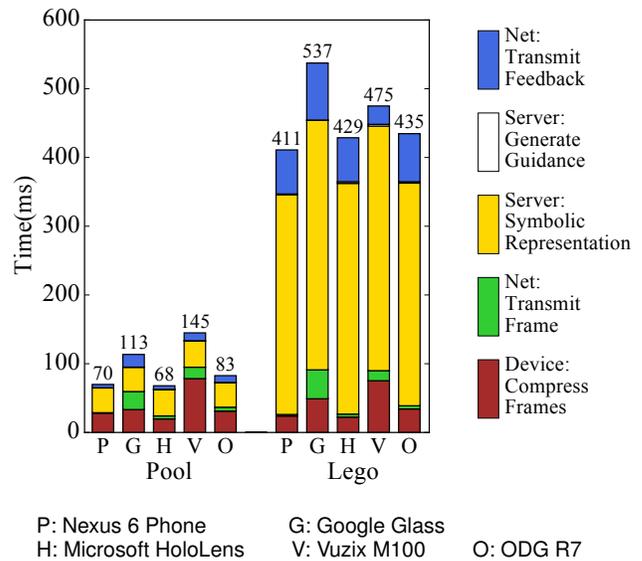


Figure 7: Latency Breakdown - Client Hardware

by 80% compared to the results with the Nexus 6 phone. Furthermore, there is greater variability in the response times, as indicated by the shallower slope of the CDFs.

To further investigate this surprising result, we capture the time breakdowns for our applications with four different client devices: the Nexus 6 phone, Google Glass, Microsoft HoloLens, Vuzix M100 Glass, and ODG R7 Glass. The measured time breakdowns are shown in Figure 7. As in Section 3.4, we only show results for two applications. For Pool, we see that the average response time (total bar height) varies significantly across client devices. A part of this difference is due to differences in processing (compression) speed on the client device. In particular, the Vuzix device takes significantly longer than the others on the compression step. The bulk of the observed differences across mobile devices can be attributed to different network transfer speeds. For example, Google Glass only has 2.4 GHz 802.11b/g technology, but the Nexus 6 phone uses 5 GHz 802.11n to fully leverage the WiFi bandwidth to minimize transfer delay. For Pool, where the network transfer delay is significant, the total response time is reduced by nearly 40% by switching from Google Glass to the Nexus 6 phone. The Vuzix device has networking that falls between Glass and the phone, while the HoloLens and the ODG device appear to be more capable wearable devices, generally matching the phone in all attributes.

Similar differences in compression and network transmission occur for Lego. However, since the server computation takes much longer here and is not affected by the choice of client hardware, the change in the overall system response time is relatively less significant than for Pool.

App	1 core	2 core	4 core	8 core
Lego	415.0	412.5	420.3	411.0
Sandwich	12579.5	7237.8	6657.6	5312.3

Figure 8: Mean Latency (ms), Varying Number of Cores

4 Reducing Server Compute Time

In the previous section, we investigated how system configuration can significantly influence the response times of the applications, primarily due to differences in network transmission and processing times on client hardware. The one aspect that remains stubbornly unchanged is the time required for processing at the server.

In this section, we look at systematic approaches to reduce the latency due to server processing. We first look at simply applying more CPU cores, and then see if hardware accelerators can help. Finally, we propose a novel technique of applying multiple algorithms in parallel with a predictor to obtain results faster with little impact on accuracy.

4.1 Leveraging More Cores

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

Can extra cores help improve the latencies achieved by our suite of applications? Here we evaluate two of the applications with large server processing components to see how their performance varies with number of virtual CPUs provided. The results are summarized in Figure 8. The Lego implementation is largely single threaded, but performs some operations that are internally parallelized in the OpenCV library. Therefore, as expected, the system response time remains almost unchanged as the number of cores is varied. The Sandwich application, on the other hand, is based on a neural network implementation, which can be executed by several well-parallelized linear algebra libraries. Therefore, it is able to run faster as the number of cores increases. However, even here, we see diminishing returns – most of the gains in latency are obtained when the number of cores is increased from one

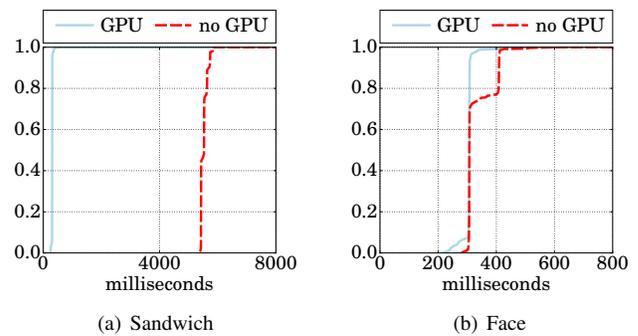


Figure 9: CDF of Latency with and without Server-side Hardware Acceleration (GPU)

to two. This is likely due to the fact that on these specific computations, the implementation of the linear algebra libraries used were not able to efficiently leverage the full parallelism provided by the system. Therefore, further increasing the core count only modestly affects latency. Even with eight cores, latency is unacceptable at about five seconds.

4.2 Leveraging Hardware Accelerators

To study the benefit of using specialized hardware, we experiment with a modern GPU as a proxy for hardware accelerators that may be available in edge computing environments of the future. We study the effect of enabling and disabling access to the GPU using Sandwich and Face, both of which use GPU-optimized neural network libraries. As Figure 9(a) shows, Sandwich benefits significantly. This is because it uses a very deep neural network, and almost all of the processing is in the accelerated code paths that leverage the GPU. On the other hand, Face gets very little benefit (Figure 9(b)), as the accelerated computations form a much smaller share of the overall computation.

As these results show, there is a large potential benefit to exposing server-side hardware accelerators to applications. However, the benefits are application-dependent, so these need to be weighed against the very real costs of this approach. Beyond the monetary costs of additional hardware (e.g., high-end GPU, or FPGA), management of a cloudlet becomes more complicated, as sharing such resources in a virtualized multi-tenant system is still a largely unsolved problem. Furthermore, applications may be tied to very particular hardware accelerators, and may or may not work well with other hardware. These compatibility issues threaten to fragment the space and limit the vision of ubiquitously available infrastructure to run any application. Finally, the relative benefits of hardware acceleration is constantly evolving as new algorithms are developed. For example, a recent work [28] uses approximation techniques to process deep neural networks at interactive speeds without hardware acceleration and very little loss in accuracy.

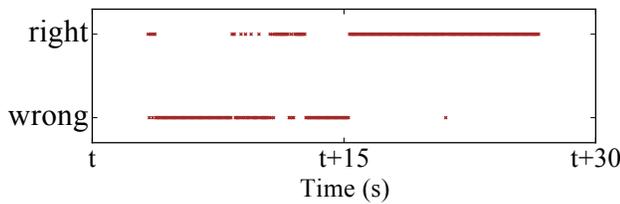


Figure 10: Temporal Locality of Algorithm Correctness with Example Object Detection Algorithm

4.3 Using Multiple Black-box Algorithms

As we have seen, simply adding cores or a hardware accelerator may not help improve latencies for our applications. In general, simply throwing additional resources at the problem has limited benefits. So what options are left for improving application latency? If one could devise a new algorithm that can solve the perception problem more efficiently, then this could reduce latency. However, hoping for such a breakthrough is not reasonable strategy. On the other hand, if one could live with a loss of accuracy, there are often relatively cheap alternative algorithms that are fast, but less accurate. Can we use the cheap algorithm to improve the performance of the system, but without suffering accuracy loss?

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

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

4.3.2 Black-box Multi-algorithm approach. In this section, we propose a novel approach for combining different algorithms to result in both high accuracy and low average latency by utilizing additional hardware. The essential idea is to run different algorithms concurrently, using the slower (but

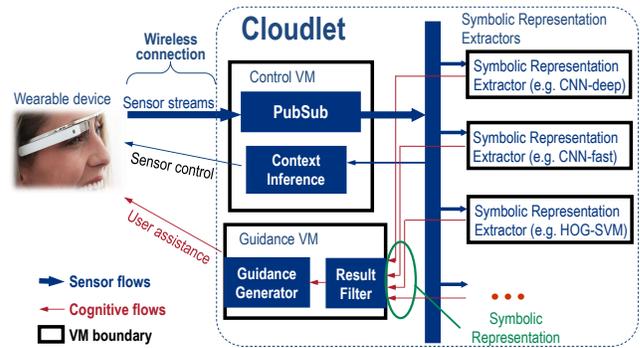


Figure 11: Adapted Gabriel Architecture for Multi-algorithm Approach to Reduce Latency

more accurate) ones to dynamically evaluate the real-time accuracy of the faster (but less accurate) ones. If a fast algorithm has been accurate for a period of time, the next result it generates will be trusted and used. If not, the system will wait for the more accurate algorithm to finish.

Why might this strategy work? The key insight behind this idea is that the mistakes an algorithm makes are not simply random events. Rather, they are correlated to the environment or the scene to which the algorithm is applied. For example, the lighting conditions, image exposure level, background clutter, and camera angle can all affect the accuracy of an algorithm. More accurate algorithms are typically more robust to changes and work well over a larger range of conditions than less accurate ones. However, because accuracy is tied to the physical scene characteristics, computer vision algorithms exhibit *temporal locality* of accuracy over sequences of video frames, as the scene conditions rarely change instantaneously. Thus, when conditions are such that the algorithm is generating accurate results, we can expect it to continue generating good results in the near term. Figure 10 demonstrates this temporal locality in an example DNN-based algorithm for object detection. Over a sequence of frames, we plot when the algorithm generates results matching known ground truth. Though often wrong, the algorithm tends to produce long runs of outputs that are all right or all wrong. This shows that temporal locality is a real phenomenon that we can exploit.

Figure 11 shows how this approach can be implemented in the Gabriel architecture. Symbolic extractors using different algorithms are run as independent instances in different Cognitive VMs. All of their results are sent to the User Guidance VM, where a result filter decides which result can be trusted. If a result is deemed accurate, it will be used for guidance generation. Otherwise, the result is simply ignored. Note that this architecture potentially allows for a black box approach, and can be applied even if the source code of algorithms are not available.

Algorithm 1 Multi-algorithm Approach

```

1: procedure PROCESS(result, algo, frame_id)
2:   if TRUST(algo, result) then
3:     Feed result to guidance generator
4:   else
5:     Mark result as useless
6:   if algo = best_algo then
7:     UPDATE_CONFIDENCE(result, frame_id)
8:   else
9:     result_history.add(frame_id, (algo, result))
10:
11: function TRUST(algo, result)
12:   if confidence[algo] ≥ THRESHOLD then
13:     return True
14:   else
15:     return False
16:
17: procedure UPDATE_CONFIDENCE(best_result, frame_id)
18:   detected_results ← result_history.get(frame_id)
19:   for (algo, result) in detected_results do
20:     if best_result = result then
21:       confidence[algo] ← confidence[algo] + 1
22:     else
23:       confidence[algo] ← 0

```

The operation of the result filter is described by Algorithm 1. Two important data structures are maintained: the *result_history* array tracks recent symbolic representation results from all algorithms, and the *confidence* array counts how many times each algorithm has been consecutively correct. The `process` procedure is called whenever a new result is generated by any of the Cognitive VMs. It uses the `trust` function to decide whether the result can be used, based on the algorithm’s recent performance. In this simple implementation, the `trust` function simply compares the associated confidence entry to a fixed threshold to decide when it should be trusted. If the result is from the best algorithm we have, we treat its result as the truth, and use it to update confidence scores of the other algorithms through the `update_confidence` procedure. Note that the accuracy can only be checked at speed of the most accurate algorithm. For example, if there are two algorithms, running at 30 and 1 FPS, then results may be produced every 33 ms when the fast one is trusted, but the confidence can be updated only once a second. If an algorithm produces a result that does not match the best one, then the corresponding confidence is set to zero.

In practice, we extend the basic Algorithm 1 to further improve accuracy. In particular, we keep a separate confidence score for each category that may be reported by a detection algorithm. This is helpful when an algorithm has different precision for different categories. In addition, we treat “nothing detected” as a special case, and only trust this result from the best algorithm. This tweak helps reduce false negatives

due to fast algorithms that are tuned to have high precision but low recall.

We demonstrate our multi-algorithm approach on three of our applications. Face, as described earlier, uses a HOG-based face detector with a DNN for accurate face recognition. We create a faster, less accurate version that uses Fisherface features [4] for recognition. A third version, in addition, uses Haar-cascade face detector [37]. For Lego, the original algorithm spends much computation compensating for lighting variations and blurriness. The faster Lego-simple algorithm removes much of this complexity, but may fail to detect the Lego shape when lighting isn’t ideal. Finally, we implemented three different Sandwich detection algorithms using different DNN architectures for object recognition. DNN1 is based on the VGG16 model [33], and can be considered state-of-the-art in terms of accuracy. DNN2 uses the ZF model [38], while DNN3 implements the VGG_CNN_F model [9].

Figure 12 shows the accuracy and latency for each of these techniques as well as our multi-algorithm approach. Clearly, our multi-algorithm approach can produce results with very low latency, but with little if any sacrifice in accuracy. Across the board, our new approach is consistently much faster than the most accurate algorithm, yet still provides significant accuracy gains over the second best algorithm. The multi-algorithm Lego even results in slightly higher accuracy than the best algorithm. This is due to the few, rare cases in which the faster algorithm is trusted and matches the ground truth, but the slower algorithm makes a mistake. Figure 13 shows the time breakdown for the multi-algorithm approach. Not surprisingly, most of the gains are due to faster symbolic representation extraction at the server. However, result transmission is also improved. This is because with the faster approach, the system operates at a higher frame rate, so device radios do not enter deep, low-power states. Thus, latency overheads of power-state transitions of the wireless network interfaces are avoided. Overall, our multi-algorithm approach demonstrates that by exploiting temporal locality, we can combine algorithms to get the best of both accuracy and speed. Furthermore, our technique shows a way to use additional processing resources to improve latency, not just throughput, of a processing task, without deep change to its algorithms.

5 When is Latency Good Enough?

So far, we have studied application latencies with different system settings, and have explored different approaches to reduce server compute time. In this section, we ask the question: are the achieved latencies good enough? To answer this question, we attempt to derive a set of target latencies for our applications. We note that this is not a precise bound, as the latencies are subject to the vagaries of human perception. Thus, our values should be treated as guidelines rather than strict limits, since human cognition exhibits high variability

Algorithms	Precision (%)	Recall (%)	Latency (ms)
DNN+HOG	99.8	96.1	333.4
Fisher+HOG	81.0	79.0	180.7
Fisher+Haar	69.4	67.5	84.6
Multi-algo	92.5	93.0	109.8

(a) Face

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

(b) Lego

Algorithms	Precision (%)	Recall (%)	Latency (ms)
DNN1	96.2	97.3	307.3
DNN2	90.1	89.5	103.0
DNN3	86.1	87.1	88.9
Multi-algo	95.4	96.7	92.5

(c) Sandwich

Figure 12: Multi-algorithm Accuracy and Mean Latency

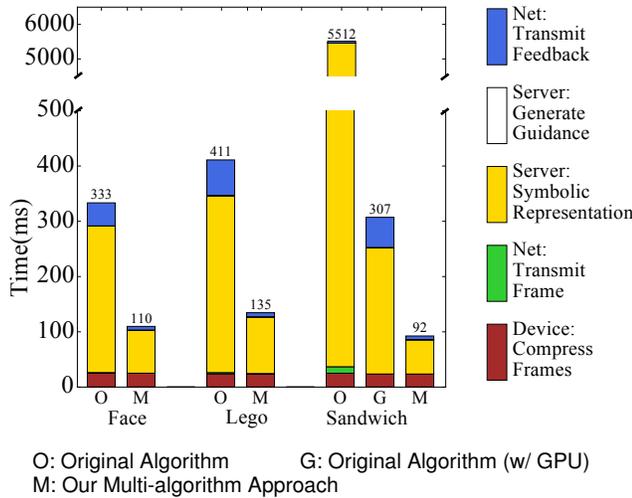


Figure 13: Multi-algorithm Latency Breakdown

due to individual differences, state of the user [25], and environmental conditions [36]. With this in mind, we derive a range of latencies for each application, with a tight and loose latency bound, to cover some of these variables.

The tight bound represents an ideal target, below which the user is insensitive to improvements, as measured, for example, by impact on performance or ratings of satisfaction. Above the loose bound, the user becomes aware of slowness, and user experience and performance is significantly impacted. Latency improvements between the two limits may be useful in reducing user fatigue, but this has not yet been validated.

5.1 Relevant Bounds in Previous Studies

Some applications have been extensively studied by previous literature, thus we use their results directly. For example, critical timing for face recognition has been published in the past. For a normal human, it takes about 370 ms to recognize a face as familiar [27], or about 1000 ms for full identity recognition [16]. If our Face application aims at providing responses at par with average human performance, it should have a target latency range of 370-1000 ms.

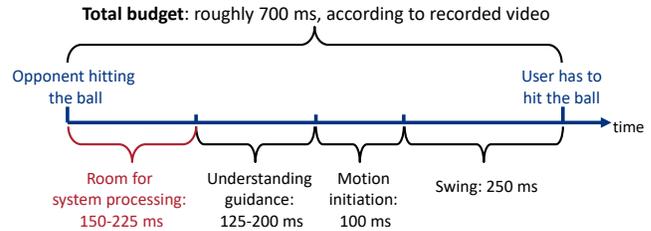


Figure 14: Deriving Latency Bounds for Ping-pong

Pool tries to provide feedback to help a user iteratively tune the cue stick position. To provide the illusion of continuous feedback, the system should react fast enough that the user perceives this as instantaneous. Miller et al. [22] indicate that users can tolerate up to 100 ms response time and still feel that a system is reacting instantaneously. He also suggested a variation range of about 10% ($\pm 5\%$) to characterize human sensitivity for such short duration of time. Hence, an upper bound of 100 ± 5 ms can be applied to Pool.

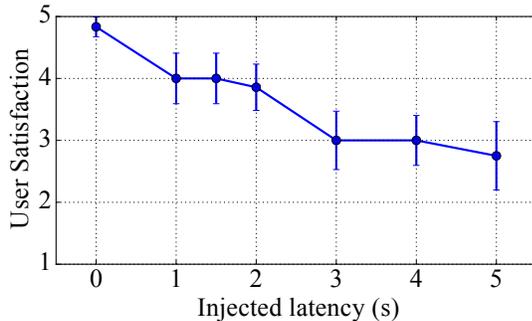
5.2 Deriving Bounds From Physical Motion

In other applications, users interact with physical systems, so the latency bounds can be derived directly from first principles of physics. Ping-pong has clear latency bound defined by the speed of the ball. From video analysis 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 time budget, the system must complete processing, deliver guidance, and leave enough time for the user to react (Figure 14). Ho and Spence [12] show that understanding a “left” or “right” audible cue requires 200 ms. Alternatively, a brief tone played to the left or right ear can be understood within 125 ms of onset [31]. In either case, we need to allow 100 ms for motion initiation by the user [19]. Our ping-pong video also suggests about 250 ms for a novice player to swing, slightly longer than the swing time of professional players [6]. This leaves 150-225 ms of the inter-hit window for system processing, depending on the audible cue.

In a similar manner, we derive the latency bound of Workout. Analysis of a video of a user doing sit-ups shows that the user rests for around 300 to 500 ms on average between the point when the completed action can be recognized and

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

Figure 15: Demographic Statistics of User Study



User satisfaction: 5=satisfied, 1=faster response desired

Figure 16: User Tolerance of System Latency

when the count information needs to be delivered (starting the next sit-up). This full 300-500 ms can be spent on system processing, but longer delays will make it likely the user will initiate a sit-up without the benefit of feedback.

5.3 Bounds for Step-by-step Tasks

Unfortunately, there are no well-defined latency bounds for Lego, Drawing, and Sandwich, which provide step-by-step instructions. Previous studies have suggested a two-second limit [22] in human-computer interaction tasks, but this was for a traditional model of interaction where a user explicitly starts an action (e.g., clicks a mouse). In contrast, for cognitive assistance applications, the user interacts with the world, and the system *infers* when a step is completed. Without an explicit point of reference, delay will likely be perceived very differently in this model. Thus, we felt it necessary to perform our own user study to investigate latency tolerance for step-by-step cognitive assistance tasks. We recruited 13 college students (demographics summarized in Figure 15) to try our Lego application using Google Glass. We asked users who were not accustomed to Glass to try some basic Glass applications before the experiments. We conducted the following two experiments.

5.3.1 Experiment 1. We first try to understand how user satisfaction changes as latency increases. We asked each user to complete four Lego-assembly tasks, each of which consisted of 7 to 9 steps. We use a "Wizard-of-Oz" approach, where a human expert takes the place of the computer vision system to determine when a step is complete, so that imperfect accuracy of the application does not affect results. Feedback

is delivered to the user after an injected latency of 0, 1, 1.5, 2, 3, 4, or 5 seconds for each task. The injected system latency for each participant during each trial was randomized, accounting for biasing in results that the order of experiments might cause. Note that the total latency experienced by the users exceeds this injected latency – we measure an additional 700 ms of latency that includes network transmission, rendering of output, and reaction time of the expert. Each participant was also asked to complete a “warm-up” trial before the four main 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. Figure 16 shows the users’ satisfaction scores for the pace of the instructions. When no latency was injected, the score was the highest, 4.8 on average. The score remains stable at around 4 with up to 2 seconds of injected delay. Beyond 2 seconds, the users were less satisfied and the score dropped below 3. These results indicate that application responses within a 2.7 seconds bound (adjusting for the additional delay in the procedure) will provide a satisfying experience to the user.

5.3.2 Experiment 2. In a second experiment, we 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 guidance both verbally and visually using printed pictures. From recordings of these interactions, we 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 [19], this suggests an ideal response time of 600 ms.

Based on these studies, we 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).

5.4 Meeting Application Latency Bounds

How close are our applications to meeting our derived bounds? Figure 17 summarizes the latency bound ranges derived for each application. For a variety of configurations, we measure and present the 90th percentile latencies achieved by the applications. The colors show when both (green), only loose (orange), or neither (red) bounds are met.

Using a WiFi-connected cloudlet has the advantage in meeting latency bounds. Six of the seven applications could meet the loose bounds, and five of them meet the tight bounds as well. Sandwich could also meet the tight bound when GPU is enabled. Offloading through LTE offers similar results, but Pool just misses its latency bounds.

	Pool	Work-out	Ping-pong	Face	Lego	Draw	Sandwich
Bound Range (tight-loose)	95-105	300-500	150-230	370-1000	600-2700		
WiFi Cloudlet w/ Phone	80	131	102	410	455	546	5708 (w/ GPU: 308)
WiFi Cloud w/ Phone	173	216	192	615	619	767	5640
LTE Cloudlet w/ Phone	107	154	123	498	578	583	5677
WiFi Cloudlet w/ Glass	144	146	164	435	618	725	5754
Using Multi-algo Approach (WiFi Cloudlet w/ Phone)				181	352		w/ GPU: 100

Figure 17: 90th Percentile System Latency vs. Latency Bounds (in milliseconds)

Using the cloud as offloading site will dramatically decrease user experience. Now, Ping-pong, Draw, and Lego fail to meet the tight bounds, and Pool substantially fails to meet even the loose bound. Using Google Glass as the client device decreases user experience to a similar degree.

Finally, using our technique of combining multiple algorithms described in Section 4.3 results in a better chance of meeting the latency bounds. Face is now able to meet the tight bound, while Lego and Sandwich now have a larger margin from the tight bounds.

6 Related Work

There have been many efforts to offer cognitive assistance through a wearable device. For example, Chroma [35] uses Google Glass to provide color adjustments for the color blind. CrossNavi [32] uses smart phone to guide blind people to cross the road. Most of these applications don't depend on computation-heavy algorithms, and can be processed entirely on a wearable device. For those applications providing advanced features using resources outside of the wearable device, a crisp response has not been a strict requirement. For example, both commercial applications like Siri or research prototypes like *Opportunity Knocks* [26] and *SenseCam* [13] could tolerate seconds of latency. In contrast, our cognitive assistance applications augment daily life in an interactive, timely manner, and cover a wider range of scenarios that have higher computation demands. We have shown that edge computing is crucial in supporting such applications.

The idea of running multiple algorithms concurrently to achieve better performance has been explored in different contexts. For example, in machine learning, the ensemble learning approach combines multiple algorithms and use methods such as voting to produce a more accurate algorithm [7]. Similarly, McFarling [21] introduced a meta algorithm to combine multiple predictors for better branch prediction. Our work also resembles the class of multiple expert meta algorithms [3] which dynamically evaluates the performance of different expert, and selects one to use at a future time. However, most existing work focus on accuracy but not

speed, and none of them have explored temporal locality of accuracy, which is the basis of our approach.

7 Conclusion

Wearable cognitive assistance may emerge as a killer application domain for edge computing. Such applications require computation offloading due to complex vision and audio processing, but are highly interactive, so need low latency access to processing resources. In this work, we implement and describe seven such applications, and carry out an empirical study to analyze their latency performance. We show that using a WiFi-connected cloudlet gives the ideal latency performance, because of the low network RTT and high bandwidth it offers. By comparing the measurement results with a set of latency bounds that we derive, we show that cloudlets can help meet user expectation for most applications. Furthermore, LTE cloudlets can also offer fast responses significantly better than using public cloud. More surprisingly, while most of the applications offload back-end processing to the cloudlet or cloud, the choice of client device also greatly impacts end-to-end latency, due to differences in networking technology.

A large portion of the end-to-end latency can be attributed to server compute time. We show that using additional CPU cores and specialized hardware accelerators could benefit a subset of applications. To further speed up computation, we have proposed a black-box multi-algorithm approach that exploits temporal locality of accuracy. By running multiple algorithms in parallel and adaptively selecting which ones to trust, we demonstrate significant speed-ups for three applications with little impact on accuracy.

Acknowledgements

This research was supported by the National Science Foundation (NSF) under grant number CNS-1518865. Additional support was provided by Intel, Google, Vodafone, Deutsche Telekom, Verizon, Crown Castle, NTT, and the Conklin Kistler family fund. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view(s) of their employers or the above-mentioned funding sources.

REFERENCES

- [1] Fractional-ball aiming. <http://billiards.colostate.edu/threads/aiming.html#fractional>. Accessed on November 27, 2015.
- [2] A. I. Anam, S. Alam, and M. 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.
- [3] S. Arora, E. Hazan, and S. Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012.
- [4] P. N. Belhumeur, J. P. Hespanha, and D. J. Kriegman. Eigenfaces vs. fisherfaces: Recognition using class specific linear projection. *IEEE Transactions on pattern analysis and machine intelligence*, 19(7):711–720, 1997.
- [5] F. Bonomi, R. Milito, J. Zhu, and S. 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*, Helsinki, Finland, 2012.
- [6] R. J. Bootsma and P. C. Van Wieringen. Timing an attacking forehand drive in table tennis. *Journal of experimental psychology: Human perception and performance*, 16(1):21, 1990.
- [7] L. Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.
- [8] G. Brown. Converging Telecom & IT in the LTE RAN. White Paper, Heavy Reading, February 2013.
- [9] K. Chatfield, K. Simonyan, A. Vedaldi, and A. Zisserman. Return of the devil in the details: Delving deep into convolutional nets. *arXiv preprint arXiv:1405.3531*, 2014.
- [10] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*, pages 68–81. ACM, 2014.
- [11] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.
- [12] C. Ho and C. Spence. Verbal interface design: Do verbal directional cues automatically orient visual spatial attention? *Computers in Human Behavior*, 22(4):733–748, 2006.
- [13] S. Hodges, E. Berry, and K. Wood. SenseCam: A wearable camera that stimulates and rehabilitates autobiographical memory. *Memory*, 19(7):685–696, 2011.
- [14] E. Iarussi, A. Bousseau, and T. Tsandilas. The drawing assistant: Automated drawing guidance and feedback from photographs. In *ACM Symposium on User Interface Software and Technology (UIST)*, 2013.
- [15] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.
- [16] M. Kampf, I. Nachson, and H. Babkoff. A serial test of the laterality of familiar face recognition. *Brain and Cognition*, 50(1):35–50, 2002.
- [17] Y. Ke, R. Sukthankar, and M. Hebert. Event detection in crowded videos. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8, 2007.
- [18] D. E. King. Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research*, 10:1755–1758, 2009.
- [19] R. L. Klatzky, P. Gershon, V. Shivaprabhu, R. Lee, B. Wu, G. Stetten, and R. H. Swendsen. A model of motor performance during surface penetration: from physics to voluntary control. *Experimental brain research*, 230(2):251–260, 2013.
- [20] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 1–14. ACM, 2010.
- [21] S. McFarling. Combining branch predictors. Technical report, Technical Report TN-36, Digital Western Research Laboratory, 1993.
- [22] R. B. Miller. Response time in man-computer conversational transactions. *December 9-11, 1968, fall joint computer conference, part I, ACM*, pages 267–277, December 1968.
- [23] OpenStack. <http://www.openstack.org/>, February 2015.
- [24] S. J. Pan and Q. Yang. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, October 2010.
- [25] R. Parasuraman and D. R. Davies. Decision theory analysis of response latencies in vigilance. *Journal of Experimental Psychology: Human Perception and Performance*, 2(4):578, 1976.
- [26] D. J. Patterson, L. Liao, K. Gajos, M. Collier, N. Livic, K. Olson, S. Wang, D. Fox, and H. Kautz. Opportunity knocks: A system to provide cognitive assistance with transportation services. In *UbiComp 2004: Ubiquitous Computing*, pages 433–450. Springer, 2004.
- [27] M. Ramon, S. Caharel, and B. Rossion. The speed of recognition of personally familiar faces. *Perception*, 40(4):437–449, 2011.
- [28] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *arXiv preprint arXiv:1603.05279*, 2016.
- [29] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.
- [30] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4), October-December 2009.
- [31] M. Schmitt, A. Postma, and E. De Haan. Interactions between exogenous auditory and visual spatial attention. *The Quarterly Journal of Experimental Psychology: Section A*, 53(1):105–130, 2000.
- [32] L. Shangguan, Z. Yang, Z. Zhou, X. Zheng, C. Wu, and Y. 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.
- [33] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [34] T. J. Tang and W. H. 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.
- [35] E. Tanuwidjaja, D. Huynh, K. Koa, C. Nguyen, C. Shao, P. Torbett, C. Emmenegger, and N. 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.
- [36] M. J. Tarr, D. Kersten, and H. H. Bülthoff. Why the visual recognition system might encode the effects of illumination. *Vision research*, 38(15):2259–2275, 1998.
- [37] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*, volume 1, pages I–I. IEEE, 2001.
- [38] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.