Energy Drain of the Object Detection Processing Pipeline for Mobile Devices: Analysis and Implications

Haoxin Wang, BaekGyu Kim, *Member, IEEE*, Jiang Xie, *Fellow, IEEE*, and Zhu Han, *Fellow, IEEE*

***Abstract*—Applying deep learning to object detection provides the capability to accurately detect and classify complex objects in the real world. However, currently, few mobile applications use deep learning because such technology is computation-intensive and energy-consuming. This article, to the best of our knowledge, presents the first detailed experimental study of a mobile augmented reality (AR) client's energy consumption and the detection latency of executing Convolutional Neural Networks (CNN) based object detection, either locally on the smartphone or remotely on an edge server. In order to accurately measure the energy consumption on the smartphone and obtain the breakdown of energy consumed by each phase of the object detection processing pipeline, we propose a new measurement strategy. Our detailed measurements refine the energy analysis of mobile AR clients and reveal several interesting perspectives regarding the energy consumption of executing CNN-based object detection. Furthermore, several insights and research opportunities are proposed based on our experimental results. These findings from our experimental study will guide the design of energy-efficient processing pipeline of CNN-based object detection.**

***Index Terms*—Object detection, augmented reality, edge computing, energy consumption, energy measurement.**

I. Introduction

With the advancement in *Deep Learning* in the past few years, we are able to create complex machine learning models for detecting objects in real-time video frames. This advancement has the potential to make Augmented Reality (AR) devices highly intelligent and enable industries to favor machine learning models with superior performance. For example, AR automotive applications (e.g., deep learning-based AR head-up-displays (HUDs)) are promised to help increase road safety, bring intuitive activities to driving, and enhance driving experience in the future. Meanwhile, as people nowadays are using their smartphones to a larger extent and also expect increasingly advanced performance from their mobile applications, the industry needs to adopt more advanced technologies to meet such expectations. One such adoption can be the use of deep learning-based AR applications.

However, few mobile AR applications use deep learning today because of inadequate infrastructure support (e.g., limited computation capacity and battery resource of smartphones). Deep learning algorithms are computation-intensive, and executed locally in ill-equipped smartphones may not provide acceptable latency for end users. For instance, in Deepmon [1], it takes approximately 600 ms for small and medium *convolutional neural network* (CNN)[1] models and almost 3 seconds for large CNN models to process one frame, which is obviously not acceptable for real-time processing [2].

Two research directions have emerged to address this challenge. The first direction is to tailor the computation-intensive deep learning algorithms to be executed on smartphones. For instance, Tiny-YOLO [3] that has only 9 convolutional layers (24 convolutional layers in a full YOLO network) is developed and optimised for use on embedded and mobile devices. TensorFlow Lite [4] is TensorFlow's lightweight solution for embedded and mobile devices. It enables low-latency inference of on-device machine learning models with a small binary size and fast performance supporting hardware acceleration. However, the reduction of the inference latency is at the cost of the precision degradation of the detection. The other research direction that is widely used in running deep learning in smartphones is to transfer all the computation data to more powerful infrastructures (e.g., the remote cloud and edge servers) and execute deep learning algorithms there [5]–[7]. Such offloading-based solutions can reduce the inference latency and extend smartphones' battery life only when the network access is reliable and sufficiently fast.

Our Motivation: Although the complexity and capabilities of smartphones continue to grow at an amazing pace, smartphones are expected to continually become lighter and slimmer. When combined with energy-hungry deep learning-based applications, the limited battery capacity allowed by

Manuscript received June 15, 2020; revised November 2, 2020 and November 19, 2020; accepted November 25, 2020. Date of publication December 1, 2020; date of current version March 18, 2021. This work was supported in part by the U.S. National Science Foundation under Grant 1718666, Grant 1731675, Grant 1910667, Grant 1910891, Grant 2025284, Grant 1839818, Grant 1717454, Grant 1731424, and Grant 1702850; and in part by Toyota Motor North America. The editor coordinating the review of this article was E. Ayanoglu. *(Corresponding author: Jiang Xie.)*

Haoxin Wang and Jiang Xie are with the Department of Electrical and Computer Engineering, University of North Carolina at Charlotte, Charlotte, NC 28223 USA (e-mail: hwang50@uncc.edu; linda.xie@uncc.edu).

BaekGyu Kim is with the Toyota Motor North America R&D, InfoTech Labs, Mountain View, CA 94043 USA (e-mail: baekgyu.kim@toyota.com).

Zhu Han is with the Department of Electrical and Computer Engineering, University of Houston, Houston, TX 77004 USA, and also with the Department of Computer Science and Engineering, Kyung Hee University, Seoul 446-701, South Korea (e-mail: hanzhu22@gmail.com).

Digital Object Identifier 10.1109/TGCN.2020.3041666

[1] A CNN is a deep learning algorithm which has demonstrated great success on image recognition, image classifications, object detection, etc.

these expectations now motivates significant investment into smartphone power management research. In order to better investigate and understand the relationship between the energy consumption and the performance of deep learning-based applications such as CNN-based object detection, we propose the following questions:

RQ 1: How is energy consumed when a CNN-based object detection application is executed locally on a mobile AR client? In order to help a mobile AR device to extend its battery life, conducting a comprehensive measurement study is significantly important.

RQ 2: Does offloading the object detection tasks to a powerful infrastructure significantly decrease both the energy consumption and latency? When a CNN-based object detection application is executed remotely, communication latency is non-negligible and unstable, especially in wireless networks. Previous work [8] shows that smartphone's radio interfaces account for up to 50% of the total power budget. In addition, improved communication speeds generally come at the cost of higher power consumption [9].

RQ 3: Besides the network condition, what else impacts the energy consumption and latency when executed remotely, and how? Executing object detection on a remote edge server is one of the most commonly used approaches to assist resource-constrained smartphones in improving their energy efficiency and performance [10]. Therefore, to further improve the efficiency for executing object detection remotely, understanding the factors that may impact the detection performance is critical.

Our Contributions: In this article, we conduct, to the best of our knowledge, the first comprehensive experimental study that investigates how a mobile AR client's energy efficiency, latency, and detection accuracy are influenced by diverse factors (e.g., CPU governor, CNN model size, and image post processing algorithm) in both local and remote executions. We make the following contributions:

1) Developing two Android benchmark applications that perform real-time object detections: one is running a light CNN model locally on the smartphone and the other is running a large CNN model remotely on an edge server.
2) Measuring and evaluating the energy consumption and latency of each phase in the implemented end-to-end CNN-based object detection processing pipeline. Both local and remote executions are investigated.
3) Comparing the local execution and the remote execution in terms of energy efficiency, latency, detection accuracy, etc.
4) Proposing several insights which can potentially guide the future design of energy-efficient mobile AR systems based on our experimental study.

The rest of this article is organized as follows. Section II discusses related work. Section III describes our proposed methodology and key performance metrics that we consider in this study. Experimental results of the local execution and remote execution are presented in Section IV and Section V, respectively. Finally, Sections VI and VII discusses threats to validity and concludes the paper, respectively.

II. Related Work

Energy Measurement: With the popularity of energy constrained mobile devices (e.g., smartphone, AR glass, and smartwatch), a number of research has investigated how the energy is consumed in mobile devices when executing applications through measurement studies [11]–[13]. Reference [14] proposes and implements a measurement framework that can physically measure the energy consumption of mobile devices and automate the reporting of measurement back to researchers. References [15], [16] study the energy consumption of GUI colors on OLED displays. In addition, the energy efficiency of network protocols such as HTTP on mobile devices has been discussed in [17], [18]. However, very few energy measurement studies focus on running deep learning-based applications on mobile devices, especially mobile AR applications. Although [19] discusses and compares the energy efficiency of different machine learning applications in terms of algorithm, implementation, and operating system (OS), our work focuses on a specific application, object detection, and conducts a comprehensive study on (i) energy efficiency comparison between local and remote executions as well as (ii) how hardware and software configurations impact the energy efficiency of executing object detections on smartphones.

Energy Modeling: Energy modeling has been widely used for investigating the factors that influence the energy consumption of mobile devices. References [20]–[24] propose energy models of WiFi and LTE data transmission with respect to the network performance metrics, such as data and retransmission rates. References [25]–[29] propose multiple power consumption models to estimate the energy consumption of mobile CPUs. Tail energy caused by different components, such as disk, Wi-Fi, 3G, and GPS in smartphones has been investigated in [11], [29]. However, none of them can be directly applied to estimate the energy consumed by mobile AR applications. This is because mobile AR applications introduce a variety of (i) energy consuming components (e.g., camera sampling and image conversion) that are not considered in the previous models and (ii) configuration variables (e.g., computation model size and camera sample rate) that also significantly influence the energy consumption of mobile devices.

CNN: In recent years, applying CNNs to object detection has been proven to achieve excellent performance [1], [3], [30]–[33]. In [34], [35], the speed and accuracy trade-offs of various modern CNN models are compared. However, none of these works considered the performance of running CNNs on smartphones. In addition, although existing papers have extensively investigated how to run CNN models on mobile devices, including model compression of CNNs [36], GPU acceleration [1], and only processing important frames [5], none of these works considered the energy consumption of executing CNNs on smartphones. In [37], a small number of measurements on the battery drain of running a CNN on a powerful smartphone are conducted. However, its battery drain results are reported by the Android OS that can only provide coarse-grained results. For example, it only shows the total battery usage of running a CNN on a smartphone

for 30 minutes. In addition, it only studies running CNNs on smartphones with high computation capabilities and the experimental results are not comparable to smartphones with poor computation capabilities.

Computation Offloading: Most existing research on computation offloading focuses on how to make offloading decisions. References [38]–[41] coordinate the scheduling of offloading requests for multiple applications to further reduce the wireless energy cost caused by the long tail problem. Reference [42] proposes an energy-efficient offloading approach for multicore-based mobile devices. Reference [43] discusses the energy efficiency of computation offloading for mobile clients in cloud computing. However, these solutions cannot be applied to improving the energy efficiency of mobile devices in mobile AR offloading cases. This is because (i) a variety of pre-processing tasks in mobile AR executions, such as camera sampling, screen rendering, and image conversion, are not taken into account and (ii) besides the latency constraint that is considered in most existing computation offloading approaches, detection accuracy is also a key performance metric, which must be considered while designing a mobile AR offloading solution. In addition, although some existing work proposes to study the tradeoffs between the mobile AR service latency and detection accuracy [35], [44], [45], none of them considered (i) the energy consumption of the mobile AR device and (ii) the whole processing pipeline of mobile AR (i.e., starting from camera sampling to obtaining detection results).

CPU Frequency Scaling: Our work is also related to CPU frequency scaling. For modern mobile devices, such as smartphones, CPU frequency and the voltage provided to the CPU can be adjusted at run-time, which is called Dynamic Voltage and Frequency Scaling (DVFS). Prior work [38], [46]–[48] proposes various DVFS strategies to reduce the mobile device energy consumption under various applications, such as video streaming [38] and delay-tolerant applications [47]. However, to the best of our knowledge, there have been no efforts factoring in the energy efficiency of mobile AR applications in the context of mobile device DVFS.

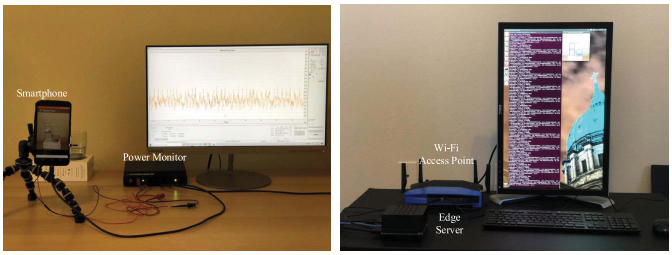
III. Proposed Methodology

This section describes the overview of our developed testbed for experimental studies, implemented benchmark applications, our proposed energy measurement process, along with the key performance metrics defined to evaluate the performance of the CNN-based object detection processing pipeline.

A. Overview of the Testbed

As shown in Fig. 1, our testbed consists of three major components: mobile AR client (e.g., smartphone), edge server attached to a WiFi access point (AP), and power monitor.

Mobile AR Client: We implement a mobile AR client on a rooted Nexus 6 smartphone running Android 5.1.1 OS. It is equipped with Qualcomm Snapdragon 805 SoC (System-on-Chip). The CPU frequency ranges from 0.3 GHz to 2.649 GHz.



(a) Mobile AR client and power monitor (b) Edge server and WiFi AP

Fig. 1. Overview of the developed testbed.

Edge Server: The edge server is developed to process received image frames sent from a smartphone and send the detection results back to the smartphone. We implement an edge server on an Nvidia Jetson AGX Xavier which is connected to a WiFi AP through a 1Gbps Ethernet cable (the length of the cable is less than 1 meter). The transmission latency between the server and AP can be ignored. Two major modules are implemented on the edge server. The first one is the communication service handler module which performs authentication and establishes a TCP socket connection with the mobile AR client. This module is also responsible for dispatching the detection results to the corresponding smartphone. The second one is the object detection module that is designed based on a custom framework called Darknet [49] with GPU acceleration and runs YOLOv3 [3], a large neural network model with 24 convolutional layers. The YOLOv3 model used in our experiments is trained on COCO dataset [50] and can detect 80 classes.

Power Monitor: To measure the power consumption, we use an external power monitor, a Monsoon Power Monitor [51], to provide power supply for the test smartphone. Different from old smartphone models, modern smartphones like Nexus 6 have very tiny battery connectors, making it very challenging to connect the power monitor to them. To solve this problem, we modify the battery connection of Nexus 6 by designing a customized circuit and soldering it to the smartphone's power input interface. In addition, the power measurements are taken with the screen on, with the Bluetooth/LTE radios disabled, and with minimal background application activity, ensuring that the smartphone's *base power* is low and does not vary unpredictably over time. For the measurements of the power consumption in local execution, the base power is defined as the power consumed by the smartphone when its WiFi interface is turned off. For the measurements of the power consumption in remote execution, the base power is defined as the power consumed when the smartphone is connected to the AP without any data transmission activity.

B. Benchmark Applications

Three benchmark applications are implemented in this article. The first application is executing CNN-based object detection on tested smartphones, defined as *local execution*. The second application is executing CNN-based object detection on our equipped edge server, defined as *remote execution*.

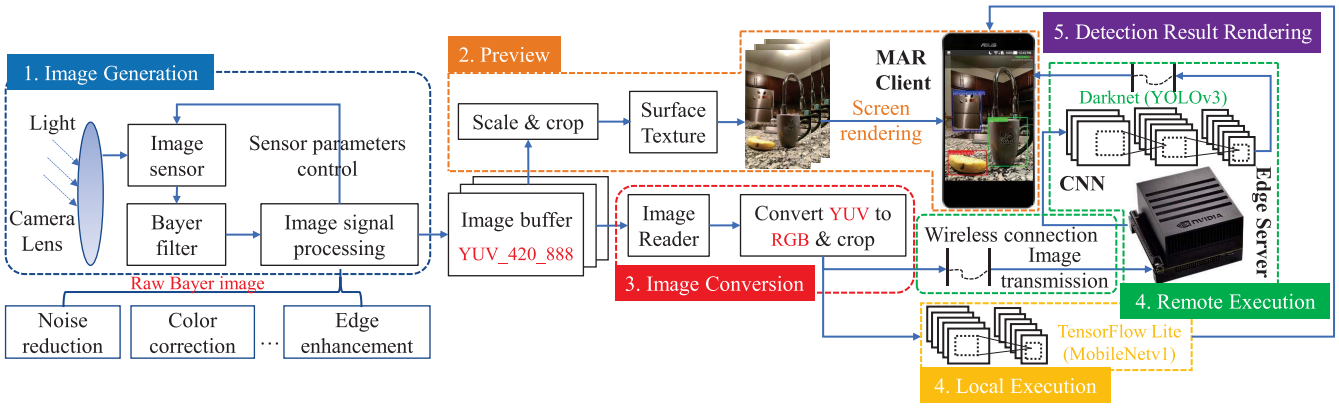
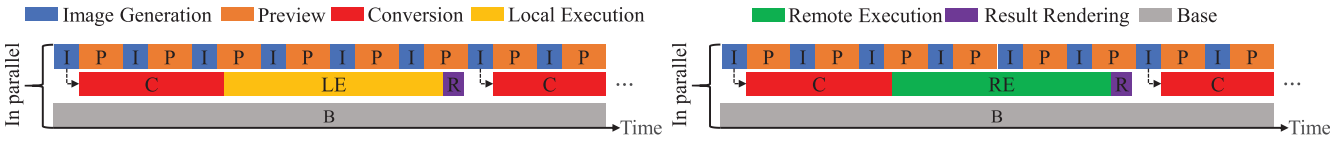


Fig. 2. Processing pipeline of the CNN-based object detection application implemented in this paper.



(a) Pipeline for the local execution

(b) Pipeline for the remote execution

Fig. 3. The diagrams of the pipelines for benchmark applications.

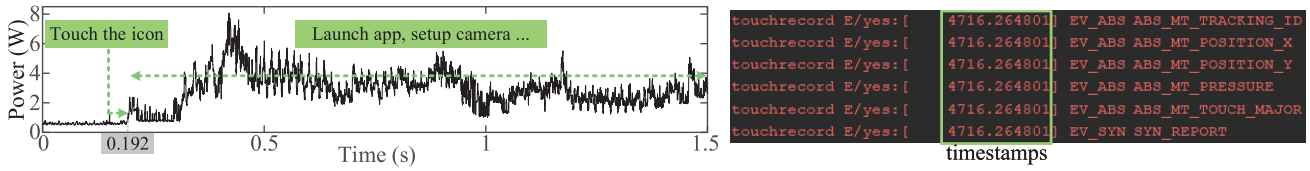
Figs. 2 and 3 provide an overview of the processing pipeline of these two benchmark applications implemented in this article, composed of five pipelined operations: 1) image generation; 2) preview; 3) image conversion; 4) local/remote execution; and 5) detection result rendering. These two benchmark applications share the same pipelined operations except Phase 4 (i.e., local execution (yellow box) and remote execution (green box)). The third application only executes the image generation and preview (i.e., Phase 1 and 2).

Image Generation (Phase 1): The input to this phase is continuous light signal and the output is an image frame. In this phase, the image sensor first senses the intensity of light and converts it into an electronic signal. A Bayer filter is responsible for determining the color information. Then, an image signal processor (ISP) takes the raw data from the image senor and converts it into a high-quality image frame. The ISP performs a series of image signal processing operations to deliver a high-quality image, such as noise reduction, color correction, and edge enhancement. In addition, the ISP conducts automated selection of key camera control values according to the environment (e.g., auto-focus (AF), auto-exposure (AE), and auto-white-balance (AWB)). The whole image generation pipeline in our benchmark applications is constructed based on `android.hardware.camera2` which is a package that provides an interface to individual camera devices connected to an Android device. `CaptureRequest` is a class in `android.hardware.camera2` that constructs the configurations for the capture hardware (sensor, lens, and flash), the processing pipeline, and the control algorithms. Therefore, in our implemented benchmark applications, we use `CaptureRequest` to set up image generation configurations. For example, `CaptureRequest.CONTROL_AE_MODE_OFF` disables AE and `CaptureRequest.CONTROL_AE_TARGET_FPS_RANGE` sets the camera FPS (i.e., the number of frames that the camera samples per second). In this article, all default image processing operations are enabled and the camera FPS is set to 15 fps.

Preview (Phase 2): The input to this phase is a latest generated image frame with `YUV_420_888` format[2] (i.e., the output of Phase 1) and the output is a camera preview rendered on a smartphone's screen with a predefined preview resolution. In this phase, the latest generated image frame is first resized to the desired preview resolution and then buffered in a `SurfaceTexture` which is a class capturing frames from an image stream (e.g., camera preview or video decode) as an OpenGL ES texture. Finally, the camera preview frame in `SurfaceTexture` is copied and sent to a dedicated drawing surface, `SurfaceView`, and rendered on the screen. In our benchmark applications, the preview resolution is set via method `SurfaceTexture.setDefaultBufferSize()`. In this article, the preview resolution is set to 800 $\times$ 600 pixels (different Android devices may have different supported preview resolution sets).

Image Conversion (Phase 3): The input to this phase is a latest generated image frame with `YUV_420_888` format (i.e., the output of Phase 1) and the output is a cropped RGB image frame. In this phase, in order to further process camera captured images (i.e., object detection), an `ImageReader` class is implemented to acquire the latest generated image frame, where `ImageReader.OnImageAvailableListener` provides a callback interface for being notified that a new generated image frame is available and method `ImageReader.acquireLatestImage()` acquires the latest image frame from the `ImageReader`'s queue while

[2]For `android.hardware.camera2`, `YUV_420_888` format is recommended for YUV output [52].



(a) Power consumption recorded by power monitor

(b) Timestamp of touch event recorded by smartphone

Fig. 4. An example of local clock synchronization.

dropping an older image. Additionally, the desired size and format of acquired image frames are configured once an `ImageReader` is created. In our benchmark applications, the desired size and the preview resolution are the same (800 $\times$ 600 pixels) and the image format in `ImageReader` is set to `YUV_420_888`. Furthermore, an image converter is implemented to convert the `YUV_420_888` image to an `RGB` image, because the input to a CNN-based object detection model must be an `RGB` image. Two image conversion methods are implemented in our benchmark applications: one is Java-based and the other is C-based (we compare these two methods in Section V-D). Finally, the converted `RGB` image is cropped to the size of the CNN model for object detections.

Local/Remote Execution (Phase 4): The input to this phase is a converted and cropped image frame (i.e., the output of Phase 3) and the output is an object detection result. In our benchmark applications, the object detection result contains one or multiple bounding boxes with labels that identify the locations and classifications of the objects in an image frame. Each bounding box consists of 5 predictions: (x, y, w, h) and a confidence score [3]. The (x, y) coordinates represent the center of the box relative to the bounds of the grid cell. The (h, w) coordinates represent the height and width of the bounding box relative to (x, y). The confidence score reflects how confident the CNN-based object detection model is on the box containing an object and also how accurate it thinks the box is what it predicts. (i) In the local execution, the benchmark application is implemented with a light framework called TensorFlow Lite [4] which is TensorFlow's lightweight solution for embedded and mobile devices. It runs a small CNN model, called MobileNetv1 [36]. In order to run MobileNetv1 with different frame resolutions in TensorFlow Lite on smartphones, we convert pre-trained MobileNetv1 SSD models to TensorFlow Lite models (i.e., optimized FlatBuffer format identified by the `.tflite` file extension). (ii) In the remote execution, the benchmark application transmits the converted and cropped image frame to the edge server through a wireless TCP socket connection in real time. To avoid having the server process stale frames, the application always sends the latest generated frame to the server and waits to receive the detection result before sending the next frame for processing.

Detection Result Rendering (Phase 5): The input to this phase is the object detection result of an image frame (i.e., the output of Phase 4) and the output is a view with overlaid augmented objects (specifically, overlaid bounding boxes and labels in this article) on top of the physical objects (e.g., a cup).

C. Energy Measurement Strategy

In order to measure the energy consumption of running those two benchmark applications on a smartphone and obtain the breakdown of energy consumed by each phase presented in Fig. 2, we design a measurement strategy. The key idea of the proposed measurement strategy is *synchronizing the recorded time in log files (saved by benchmark applications in the tested Android smartphone) and power measurement data (exported by the Monsoon power monitor)*. However, this is very challenging, because the tested smartphone and the power monitor do not share the same global clock. For example, in Android smartphones, the recorded time of an event can be counted by a system clock, `uptimeMillis`,[3] where the clock is counted in milliseconds since the system is booted (e.g., if an event happens 100 milliseconds after the system is booted, the exported timestamp of the event in the log file is 100). On the other hand, in the power monitor, the timestamp is counted in milliseconds since the power measurement is launched.

Local Clock Synchronization & Event Localization: To synchronize the exported timestamps of the Android smartphone and the power monitor, we propose to set up a *flag event* that can be tracked easily and accurately in both of them. The touch event that launches the benchmark application is selected as the flag event to synchronize the timestamps. For example, Fig. 4(a) illustrates the power consumption of the tested smartphone recorded by a Monsoon power monitor. The power measurement is launched at time 0 and the smartphone only consumes the base power, described in Section III-A. Then, we touch the icon of the benchmark application at time 0.192s (i.e., the moment that the benchmark application is launched). On the other hand, Fig. 4(b) depicts the timestamps recorded by the Android kernel[4] when the touch event is triggered, which denotes that the touch event happens at time 4716.264801s. After the timestamp of the flag event is acquired, the local clocks in the tested smartphone and the power monitor can be synchronized easily and accurately. For example, if the start and end time for executing an image conversion are recorded as 4726.136s and 4726.612s in the log file generated by the benchmark application, the power consumption of the image conversion can be localized between

[3]This clock stops when the system enters deep sleep (CPU off, display dark, and device waiting for external input), but is not affected by clock scaling, idle, or other power saving mechanisms. Additionally, it is guaranteed to be monotonic, and is suitable for interval timing when the interval does not span device sleep.

[4]These timestamps are also generated by clock `uptimeMillis` but with microsecond precision.

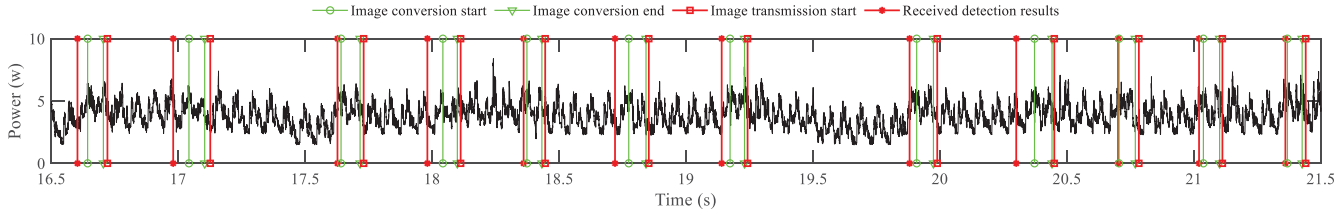


Fig. 5. An example of event localization in the collected power measurement data (CPU governor: Interactive, remote execution with C-based image conversion method).

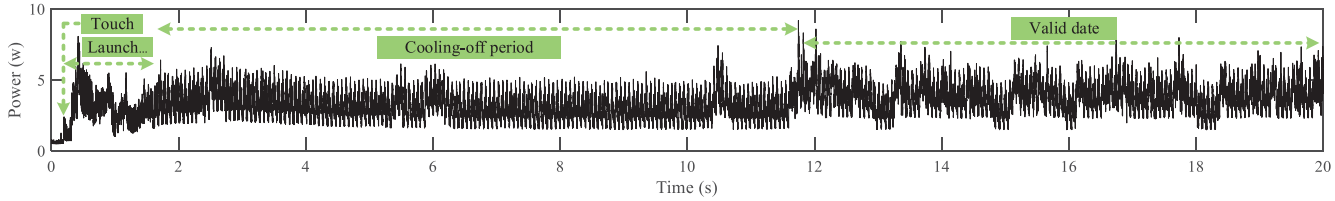


Fig. 6. Power measurement and valid data collection process.

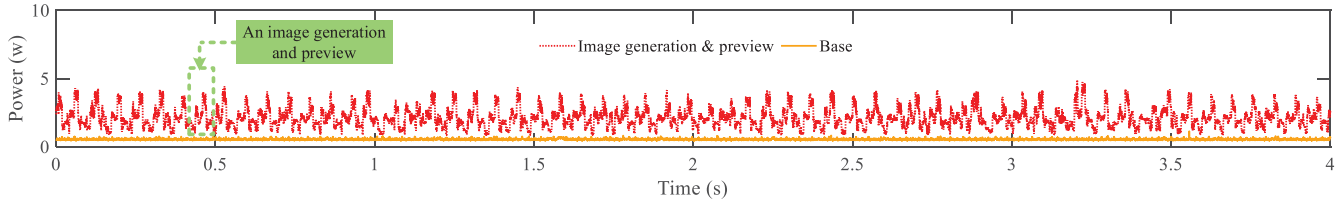


Fig. 7. A fragment of the power consumption of image generation and preview and base (CPU governor: Interactive).

10.063s and 10.539s in the power measurement data recorded by the power monitor. Fig. 5 presents an example of event localization in the collected power measurement data through our proposed strategy.

Script for Capturing the Touch Event: Since the touch event happens before the benchmark application launching, the function of capturing the touch event cannot be directly added into the benchmark application. In addition, the tested smartphone's USB interface is automatically disabled when the power measurement starts. Thus, the smartphone cannot be instructed to start or terminate a touch event listener through transmitting `adb shell` commands by a computer. Considering the above mentioned limitations, we design a lightweight application and implement it to record touch events. The function of this application is to run a touch event listener in the background using an `adb shell` command `getevent -lt /dev/input/event0`. We evaluate whether running this background touch event listener will impact the power consumption of benchmark applications. The measurement results show that the average power consumption of the smartphone is 3.721W (running the remote execution benchmark application with the touch event listener) and 3.704W (running the remote execution benchmark application without the touch event listener), where the two measurements are under the same conditions and each measurement runs for 5 minutes. Therefore, the result demonstrates that our background touch event listener has little impact on the power consumption of benchmark applications.

Cooling-off Period: Furthermore, in order to mitigate the interference from screen touching, application launching, camera initializing, and CNN model loading in the collected power measurement data, a cooling-off period is set up, as shown in Fig. 6. In the cooling-off period, benchmark applications only executes Phases 1 and 2 for generating a fixed number of image frames (e.g., 150 frames in this article). After the cooling-off period, benchmark applications start executing the whole processing pipeline and generating valid power consumption data.

Power/Energy Consumption Dissection: Fig. 3 illustrate that image generation and preview (Phases 1 and 2), image conversion and local/remote execution (Phases 3 and 4), and base (e.g., OS and screen) are executed in parallel. The workload of running our benchmark applications on the tested smartphone is composed of these three parallel executions. In addition, the power consumption is increased by the workload increment. Therefore, our strategy for dissecting the power consumption of each phase is:

1) Measuring the power consumption of the whole processing pipeline, image generation and preview (Phases 1 and 2),[5] and base separately with the same configurations (e.g., CPU governor, camera sampling rate, and preview resolution) and conditions (e.g., background activity and screen brightness). Figs. 5 and 7 present examples of how the power consumption of these three parallel executions look like.

[5]In order to measure the power consumption of phases 1 and 2, we implement an application that only executes image generation and preview, where it uses the same Android camera package (i.e., `android.hardware.camera2`) and camera configurations (e.g., preview resolution) with our benchmark applications.

2) Isolating the power consumption of (i) image generation and preview + image conversion + base, (ii) image generation and preview + local/remote execution + base, and (iii) image generation and preview + others + base through the proposed clock synchronization and event localization strategy.
3) Obtaining the power consumption of image conversion, local/remote execution, and others by subtracting the average power consumption of image generation and preview and base from cases (i), (ii), and (iii), respectively.
4) Obtaining the energy consumption of each phase via calculating the integral of the power consumption over the latency. For example, the energy consumption of an image conversion is the sum of its power consumption within an image conversion latency.

Validation: Paper [13] observed that a single energy measurement could be misleading due to the variability in energy consumption. Therefore, in this article, all of our measurement experiments are repeated multiple times, and each energy consumption and latency result shown in Sections IV and V is the mean value of completing 200 object detections. Since we observe that the variance of the mean value of the measured data, such as power consumption, per frame latency, and per frame energy consumption, is negligible after the number of the collected object detections is over 200 in all of our measurement experiments, collecting measurement results based on 200 object detection executions is good enough for achieving stable and accurate results. Furthermore, in order to ensure that each measurement is launched with a clean environment, the benchmark application is re-installed on the tested smartphone through Android Studio and the data generated during the execution such as log files are transferred to a workstation and removed from the smartphone after each measurement, even though the configuration of the benchmark application does not require to be changed in the next measurement.

D. Key Performance Metrics

We define three performance metrics to evaluate the performance of the CNN-based object detection processing pipeline implemented in this article:

Per Frame Latency: The per frame latency is the total time needed to obtain the detection results on one image frame (i.e., usually shown as one or multiple bounding boxes that identify the locations and classifications of the objects in a frame). In this article, it is defined as the time period from the moment the *Image Reader* acquires one camera captured image frame to the moment the bounding boxes are drawn on the mobile AR client's screen, as depicted in Fig. 2. In the local execution, the per frame latency includes the time used for converting the YUV frame to the RGB frame, cropping the frame to the fitted resolution $k \times k$ pixels, and executing CNN, defined as *inference latency*, on the smartphone. In the remote execution, the per frame latency includes, besides the image conversion and crop latency that are both executed locally on the smartphone, the communication latency (i.e., transmitting the frame and receiving the results) and the inference latency on the edge server.

Per Frame Energy Consumption: The per frame energy consumption is the total amount of energy consumed in a mobile AR client by successfully performing the object detection on one image frame. In the local execution, the per frame energy consumption includes the energy consumed by camera sampling (i.e., image generation), screen rendering (i.e., preview), image conversion, inference, and operating system (i.e., base). In the remote execution, it includes the energy consumed by camera sampling, screen rendering, image conversion, communication, and operating system. In a per frame energy consumption, the image generation and preview are usually executed multiple times (depends on the length of the per frame latency), while the image conversion and local/remote execution are executed only once.

Detection Accuracy: The mean average precision (mAP) is a commonly used performance metric in object detection. Better performance is indicated by a higher mAP value. Specifically, the average precision [53] is computed as the area under the precision/recall curve through numerical integration. The mAP is the mean of the average precision across all classes.

IV. Experimental Results of Local Execution

RQ 1: How is energy consumed when a CNN-based object detection application is executed locally on a mobile AR client? To answer this question, in this section, we describe our efforts towards measuring and understanding the energy consumption and the performance of running CNN models on smartphones locally. We begin by measuring the per frame latency and the per frame energy consumption of executing CNN-based object detection under different smartphone's CPU governors in Section IV-A. In addition, we explore the impact of the CNN model size on the per frame latency and the per frame energy consumption in Section IV-B. Lastly, in Section IV-C, we summarize the insights from our measurement studies and discuss potential research opportunities for improving the energy efficiency of locally executing CNN-based object detection on smartphones.

A. The Impact of CPU Governor

CPU Governor[6]*:* Dynamic voltage and frequency scaling (DVFS) is a technique commonly used for dynamically adjusting the voltage and frequency of a mobile device's CPU in order to balance the trade-off between the power consumption of the device and the required performance. In order to offer DVFS, the CPU provides a set of valid voltages and frequencies that can be dynamically selected by a power management policy which is usually called a CPU governor. Different CPU governors adjust the CPU voltage and frequency based on variant criteria such as CPU usage. The six most popular CPU governors are described as follows:

- *Conservative governor:* It adjusts the CPU frequency based on the current usage and it biases the mobile device

[6]We change the Android smartphone's CPU governor manually by writing files in `/sys/devices/system/cpu/[cpu#]/cpufreq/ scaling_governor` virtual file system with root privilege.

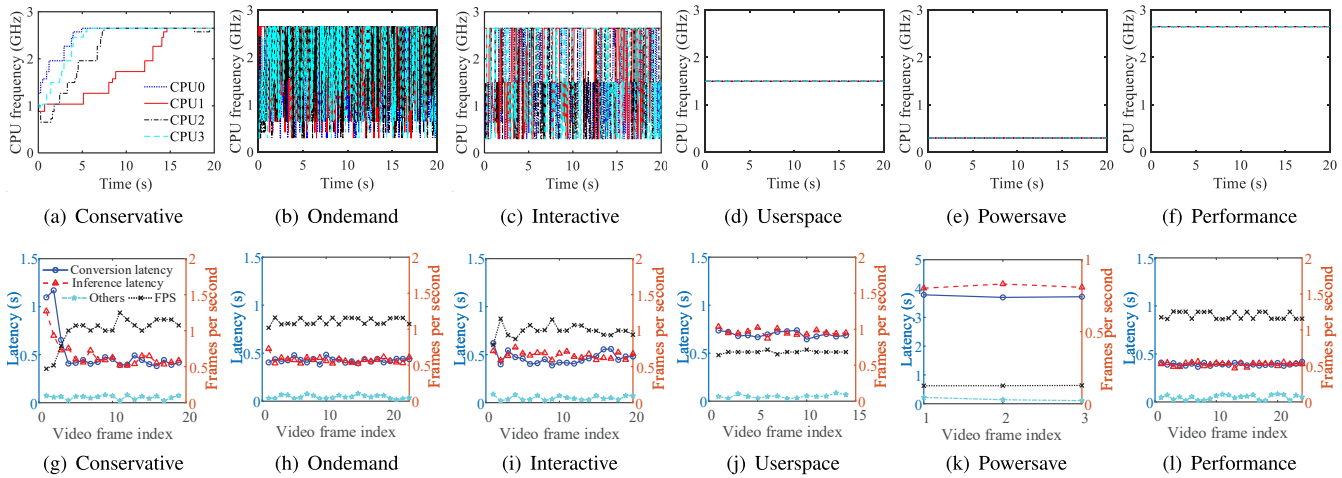


Fig. 8. CPU governor vs. per frame latency (CNN model size: 300 × 300 pixels).

to prefer the lowest possible CPU frequency as often as possible. In other words, a large and persistent load can be placed on the CPU only before the CPU frequency is raised. Thus, the conservative governor is good for the mobile device's battery life.

- *Ondemand governor:* It adjusts the CPU frequency based on the current usage, which is similar to the conservative governor. However, the ondemand governor immediately boosts the CPU to the highest possible frequency when there is a load on the CPU and switches back to the lowest possible frequency when the CPU is idle rather than gradually increases and decreases the CPU frequency. Thus, it offers excellent interface fluidity due to its high-frequency bias.
- *Interactive governor:* It is the default CPU governor for most android mobile devices. Similar to conservative and ondemand governors, it sets the CPU frequency based on the current usage. However, the interactive governor is designed for latency-sensitive and interactive workloads, so it is more aggressive about scaling the CPU speed up in response to CPU-intensive activities.
- *Userspace governor:* It allows the user or any userspace program to set the CPU to a specific frequency (i.e., the CPU frequency is set to 1.497 GHz in this work), whereas it only allows the CPU frequency to be set to predefined fixed values.
- *Powersave governor:* It sets the CPU statically to the lowest possible frequency to minimize the energy consumption of the mobile device's CPU.
- *Performance governor:* It sets the CPU statically to the highest possible frequency to maximize the performance of the mobile device's CPU.

Per Frame Latency: We first seek to investigate how the CPU governor impacts the per frame latency of object detection in the local execution scenario, where a CNN model is executed on a smartphone and the model size is 300 × 300 pixels. The experimental results are shown in Fig. 8, where Figs. 8(a)–8(f) depict the frequency variations of the tested smartphone's CPUs and Figs. 8(g)–8(l) illustrate the latency of each phase in the object detection processing pipeline. We show the latency of the two highest time-consuming phases, image conversion and inference latency, which comes up to 95% of the per frame latency. We observe that (1) as described above, the conservative governor provides a graceful CPU frequency increase, which causes temporarily high per frame latency and low frame per second (FPS) when the object detection application is launched, as shown in Figs. 8(a) and 8(g). This observation demonstrates that the conservative governor may not be suitable for CNN-based object detection applications because object detection requires a high fluidity to interact with the user. (2) Although both ondemand and interactive governors provide aggressive responses to the execution of object detection, as depicted in Figs. 8(b) and 8(c), the ondemand governor offers a relatively steadier latency performance than the interactive governor due to its high-frequency bias. In Figs. 8(d), 8(e), and 8(f), the CPU is set to a user-defined, the lowest, and the highest possible frequencies, respectively. (3) It is not surprising to find that the powersave governor is the worst-performing governor in terms of the latency, where its per frame latency is almost eight times higher than that of the performance governor. (4) The performance governor outperforms other presented CPU governors in terms of latency, and the measured average per frame latency is shown in Table I.

Per Frame Energy Consumption: We next examine how the CPU governor impacts the per frame energy consumption of executing object detection on the smartphone. The experimental results are shown in Fig. 9, where Figs. 9(a)–9(f) depict the power consumption; Figs. 9(g)–9(l) illustrate the average per frame energy consumption; and Figs. 9(m)–9(r) depict the average percentage breakdown of energy consumed by each phase in the processing pipeline. We make the following observations. (5) The performance governor consumes the highest power consumption, as shown in Fig. 9(f), because the processors always run with the highest possible CPU frequency. Although it is capable of providing the best latency performance, continuously run with the highest CPU frequency may cause the smartphone overheating and trigger

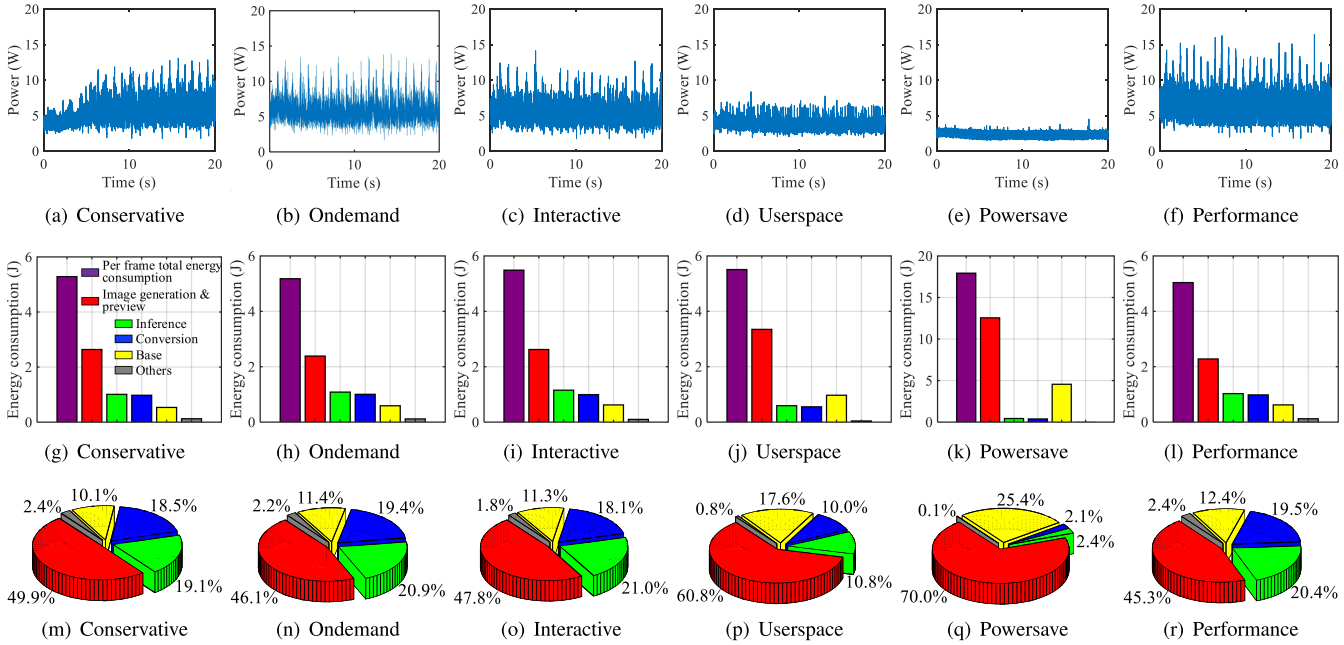


Fig. 9. CPU governor vs. power and average per frame energy consumption (CNN model size: 300 × 300 pixels).

TABLE I
LATENCY RESULTS OF THE LOCAL EXECUTION WITH DIFFERENT CPU GOVERNORS

CPU Governor	Conservative	Ondemand	Interactive	Userspace	Powersave	Performance
Per Frame Latency (second)	0.915	0.904	1.013	1.444	7.766	**0.823**
Image Conversion Latency (second)	0.436	0.425	0.466	0.693	3.713	**0.391**
Inference Latency (Second)	0.424	0.431	0.501	0.699	3.971	**0.386**
Others (Second)	0.055	0.047	0.047	0.052	0.082	**0.046**

TABLE II
PER FRAME ENERGY CONSUMPTION RESULTS OF THE LOCAL EXECUTION WITH DIFFERENT CPU GOVERNORS

CPU Governor	Conservative	Ondemand	Interactive	Userspace	Powersave	Performance
Power Consumption (watt)	5.357	5.725	5.415	3.814	**2.308**	6.115
Per Frame Energy Consumption (Joule)	5.284	5.179	5.487	5.508	17.926	**5.037**
Image Generation & Preview Energy Consumption (Joule)	2.639	2.385	2.622	3.349	12.552	**2.281**
Inference Energy Consumption (Joule)	1.009	1.084	1.153	0.593	**0.432**	1.028
Image Conversion Energy Consumption (Joule)	0.977	1.004	0.992	0.553	**0.380**	0.983
Base Energy Consumption (Joule)	**0.534**	0.591	0.621	0.971	4.555	0.622
Others (Joule)	0.125	0.115	0.099	0.042	**0.007**	0.123

CPU throttling mechanisms to avoid thermal emergencies by sacrificing the performance. (6) Interestingly, the performance governor provides the lowest per frame energy consumption, while the powersave governor offers the highest per frame energy consumption, as shown in Table II. *This observation indicates a critical trade-off between the battery life (i.e., power consumption) and per frame energy consumption in CNN-based object detection applications.*

In order to dissect the energy drain through different processing pipeline phases, we break down the per frame energy consumption as follows: image generation and preview, inference, image conversion, base, and others. We find that (7) the image generation and preview phase always contributes the highest energy consumption (i.e., approximately 45.3% - 70.0%). The reason it consumes considerably high energy is executing the 3A (i.e., AF, AE, and AWB) and multiple fine-grained image post processing algorithms (e.g., noise reduction (NR), color correction (CC), and edge enhancement (EE)) on ISP. These sophisticated algorithms are designed to make an image that is captured by the smartphone camera look perfect. *However, is it always necessary for the camera captured frame to be processed by all of those energy-hungry image processing algorithms in order to achieve a successful object detection result?* In addition, the number of frames captured by the camera per second is a fixed value (e.g., 24 or 30 frames/second) or in a range (e.g., [7, 30] frames/second), which is controlled by the AE algorithm. However, due to the limited computation capacity of smartphones, usually the detection FPS is far slower than the camera capture frame rate. On the other hand, the CNN always extracts the latest camera captured frame, which indicates that, from the perspective of the energy efficiency of the object detection pipeline, capturing frames with a fast rate is unnecessary and energy-inefficient. *Therefore, both raising CPU frequency and decreasing camera capture frame*

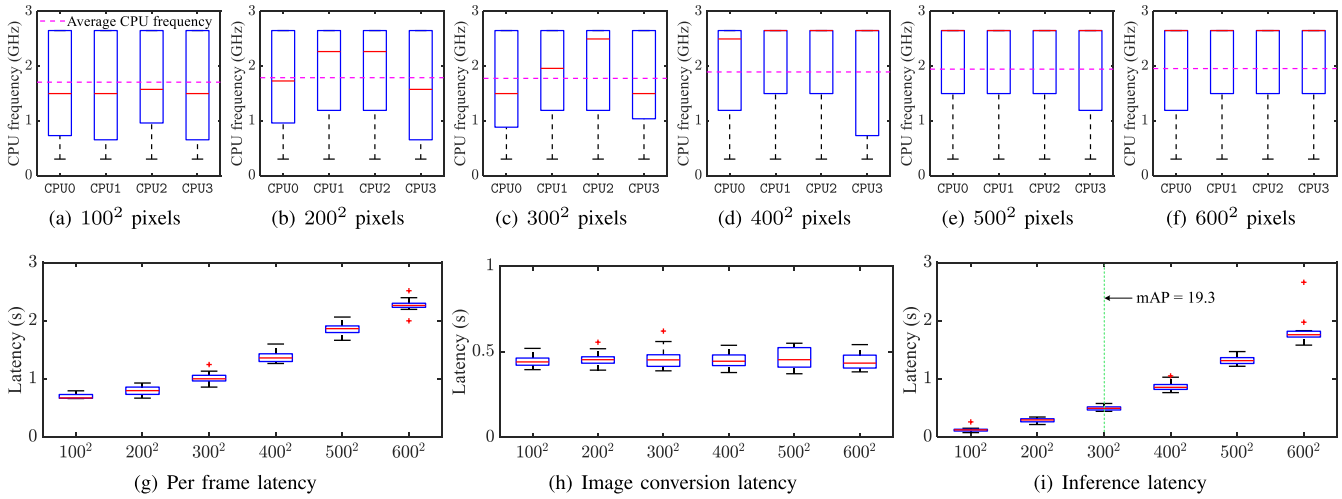


Fig. 10. CNN model size vs. CPU frequency & latency (CPU governor: interactive).

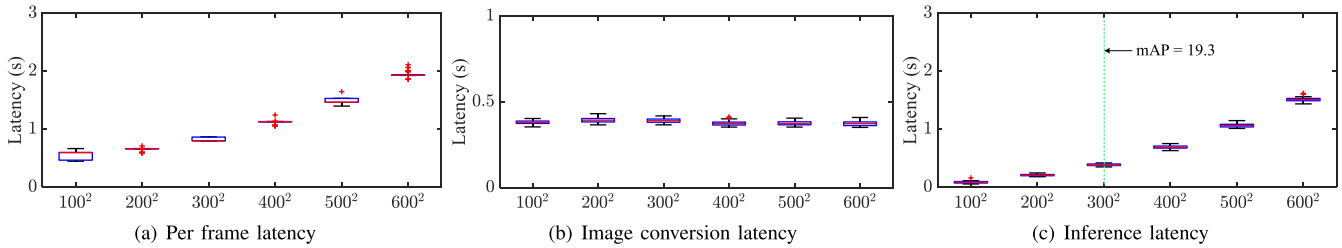


Fig. 11. CNN model size vs. latency (CPU governor: performance).

rate are efficient approaches to reduce the energy consumption of image generation and preview.

Besides the energy consumption of image generation and preview, inference and image conversion phases consume a large amount of energy, as depicted in Fig. 9 and Table II. (8) Although a low CPU frequency incurs high per frame energy consumption, it decreases the energy consumption of both inference and image conversion phases. This observation indicates that *there is a trade-off between the energy consumption reduction of image generation and preview phases and inference and image conversion phases.* For example, raising the CPU frequency can decrease the energy consumption of image generation and preview phases but concurrently increases the energy consumption of inference and image conversion phases. Furthermore, (9) the conservative governor provides the lowest base energy consumption, which demonstrates that existing CPU governors are capable of scaling CPU frequency for the workload of the smartphone's operating system.

B. The Impact of CNN Model Size

CNN Model Size: Recently, CNN-based methods have become the leading approach for achieving high quality object detection. The CNN model size determines the detection accuracy (i.e., mAP). Increasing the CNN model size always results in a gain of mAP [54], [55]. In this section, we seek to investigate how the CNN model size impacts the per frame latency and energy consumption of executing the object detection on the smartphone.

Per Frame Latency: In this experiment, we implement the MobileNets [36] with six different CNN model sizes (i.e., from 100 $\times$ 100 to 600 $\times$ 600 pixels). Figs. 10 and 11 depict the latency results of running CNN-based object detection with different model sizes, where the smartphone works on interactive and performance governors, respectively. We make the following observations. (10) Running a large CNN model increases the average CPU frequency under the interactive governor, as shown in Figs. 10(a)–10(f). This observation demonstrates that a larger CNN model will generate more workload on the smartphone's CPU. (11) A larger CNN model always results in a higher per frame latency for both interactive and performance governors, as depicted in Figs. 10(g) and 11(a), where the per frame latency of the interactive and performance boosts 220% and 247%, respectively, when the CNN model size increases from 100 $\times$ 100 to 600 $\times$ 600 pixels. (12) The per frame latency increment is mainly from the raise of the inference latency, while the image conversion latency does not vary much when the CNN model size increases, as illustrated in Figs. 10(h), 10(i), 11(b), and 11(c). This is because no matter what the CNN model size $k \times k$ is configured, every YUV frame is converted to an RGB frame with the preview resolution $k_1 \times k_2$ first. After the image conversion is completed, the RGB frame is resized to $k \times k$ pixels. (13) For each CNN model size, the performance governor provides a lower per frame latency (i.e., 14%-21%) than the interactive governor, which demonstrates that our observation (4) can be applied to diverse CNN model sizes.

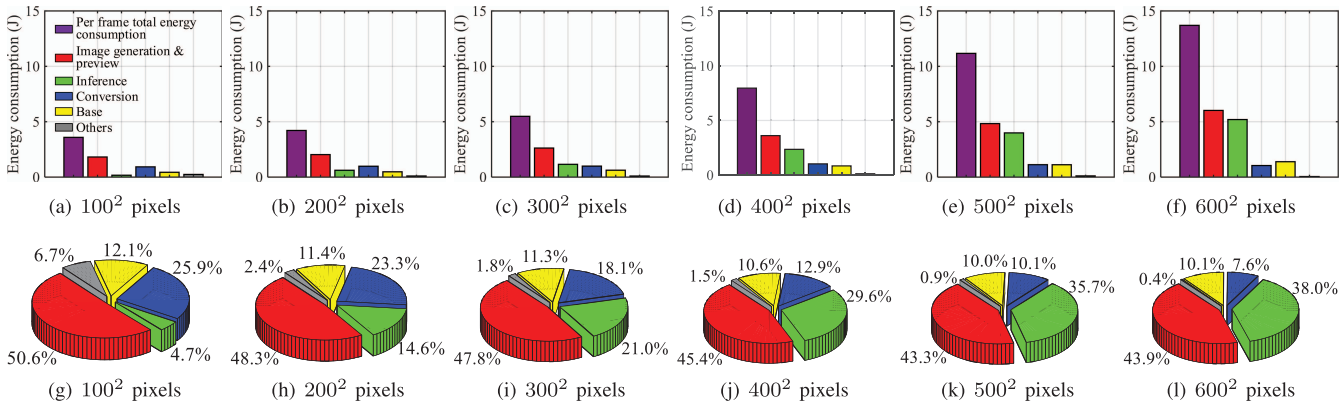


Fig. 12. CNN model size vs. per frame energy consumption (CPU governor: interactive).

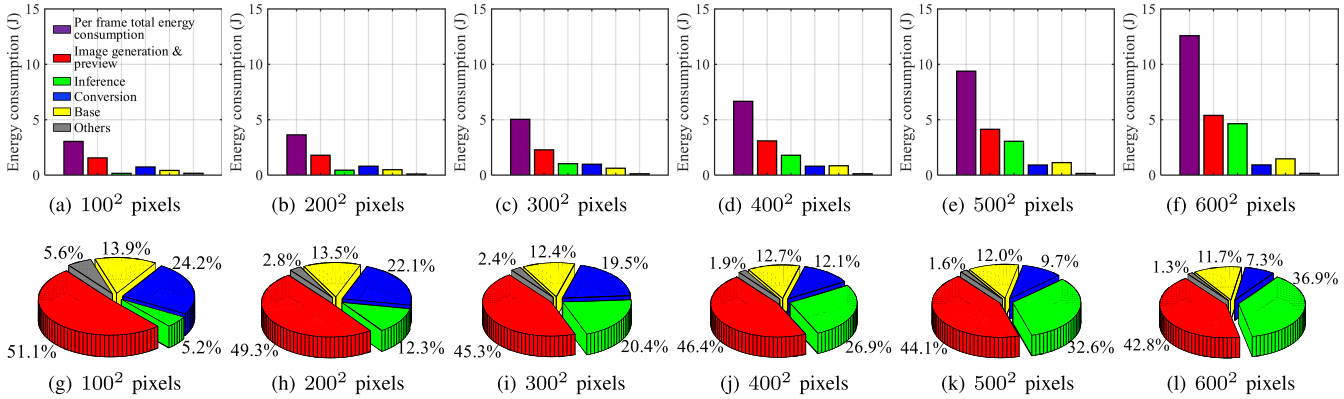


Fig. 13. CNN model size vs. per frame energy consumption (CPU governor: performance).

Per Frame Energy Consumption: We next examine how the CNN model size impacts the per frame energy consumption of executing object detection on the smartphone. Figs. 12 and 13 depict the measured per frame energy consumption results of running CNN-based object detection with different model sizes, where the smartphone works on interactive and performance governors, respectively. Figs. 12(a)–12(f) and 13(a)–13(f) illustrate the average per frame energy consumption; and Figs. 12(g)–12(l) and Figs. 13(g)–13(l) depict the average percentage breakdown of energy consumed by each phase in the processing pipeline. We observe that (14) the per frame energy consumption grows dramatically as the CNN model size increases, which is mainly contributed by the inference energy consumption increment. For example, the inference energy consumption accounts for 4.7% and 38.0% of the per frame energy consumption when the CNN model size is 100 $\times$ 100 and 600 $\times$ 600 pixels, respectively, as shown in Fig. 12. In addition, although increasing the CNN model size always results in a gain of mAP, the gain of mAP becomes smaller as the increase of the model size [3]. *This observation inspires us to trade mAP for the per frame energy consumption reduction when the CNN model size is large.* (15) There is a reduction in the proportion of the energy consumption of both the image generation and preview phase and base phase when the CNN model size grows. As we discussed in Section IV-A, a large proportion of the image generation and preview energy consumption to the per frame energy consumption may result in the smartphone expending significant reactive energy for sampling non-detectable image frames. These two observations indicate that there is a trade-off between reducing the per frame energy consumption and decreasing the proportion of the reactive energy. Therefore, a comprehensive approach for improving the energy efficiency of executing CNN-based object detection on smartphones must take into account the reduction of both the per frame energy consumption and the proportion of the reactive energy.

C. Insights and Research Opportunities

Insights:

- Ondemand and performance CPU governors achieve lower per frame energy consumption and latency than the other four popular CPU governors when the smartphone locally executes the CNN-based object detection application. However, as the smartphone's CPUs keep running at the highest frequency in the performance governor, it may cause the smartphone overheating and trigger CPU throttling mechanism to avoid thermal emergencies by sacrificing the performance. Therefore, the ondemand governor is recommended as the default CPU governor of the local execution, which supports sustainable and low per frame energy consumption and latency object detections.

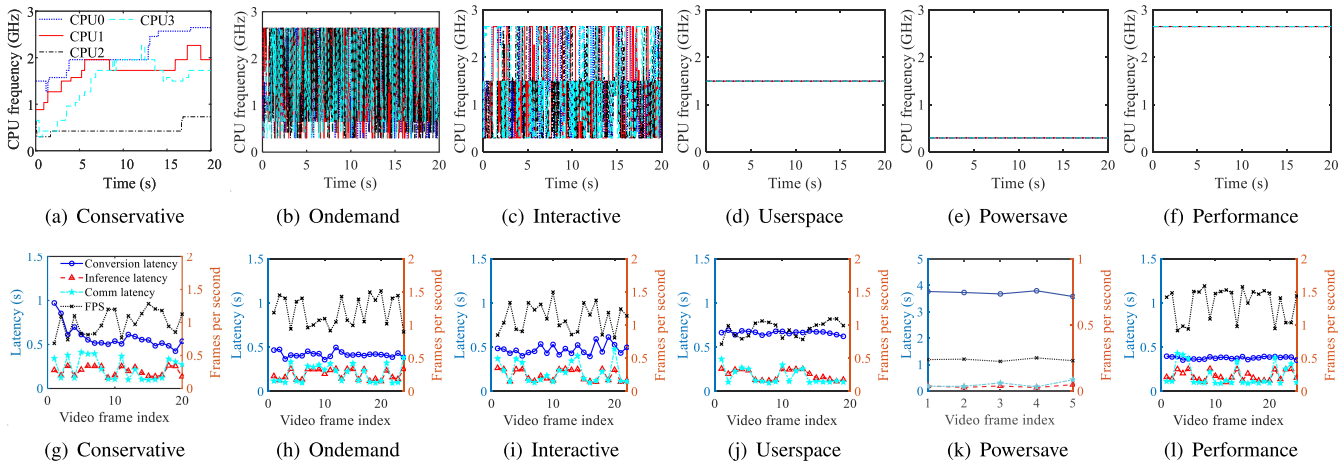


Fig. 14. CPU governor vs. per frame latency (CNN model size: 320 × 320 pixels).

- Both the CPU governor (i.e., CPU frequency) and the CNN model size significantly impact the per frame latency and energy consumption. However, simply increasing the CPU frequency or decreasing the CNN model size is inadequate to minimize the per frame energy consumption because different phases may have opposite reactions.
- Increasing the CNN model size always results in a gain of mAP and an increment of the per frame energy consumption. However, the amount of the increment of mAP becomes smaller as the increase of the model size, while the increment of the per frame energy consumption becomes larger as the increase of the model size. Therefore, this observation inspires us to trade mAP for the per frame energy consumption reduction when the CNN model size is large.
- In order to improve the energy efficiency of smartphones that locally execute the CNN-based object detection, we must jointly consider the per frame energy consumption, the proportion of the reactive energy, and the battery life.

Research Opportunities:

- Current CPU governors cannot achieve energy-efficient object detection on smartphones (i.e., jointly considering the per frame energy consumption, the proportion of the reactive energy, and the battery life). A CPU governor specifically designed for CNN-based object detection applications is critical and desirable.
- An intelligent configuration adaption algorithm that is capable of selecting the best combination of the CPU governor, CNN model size, and camera sample rate according to the smartphone's battery life, processor's temperature, and detection accuracy requirement might be a potential solution for achieving energy-efficient and high-performance object detection.

V. Experimental Results of Remote Execution

RQ 2: Does offloading the object detection tasks to a powerful infrastructure significantly decrease both the energy consumption and latency? To answer this question, in this section, we describe the experimental results on evaluating the impact of various factors on the energy consumption of a mobile AR client, latency, and detection accuracy of remotely executing CNN-based object detection on smartphones. We begin by measuring the per frame latency and the per frame energy consumption of executing CNN-based object detection under different smartphone's CPU governors in Section V-A. In addition, we explore the impact of the CNN model size on the per frame latency and the per frame energy consumption in Section V-B. Furthermore, the image generation and preview phase and image conversion phase are discussed in Sections V-C and V-D, respectively. Lastly, in Section V-E, we summarize the insights from our measurement studies and discuss potential research opportunities for improving the energy efficiency of remotely executing CNN-based object detection on smartphones.

A. The Impact of CPU Governor

Per Frame Latency: We first seek to investigate how the CPU governor impacts the per frame latency of object detection in the remote execution scenario, where a CNN model is executed on the implemented edge server with a 5GHz WiFi link to the smartphone. The executed CNN model size is 320 × 320 pixels. The experimental results are shown in Fig. 14, where Figs. 14(a)–14(f) depict the frequency variations of the tested smartphone's CPUs and Figs. 14(g)–14(l) illustrate the latency of each phase in the object detection processing pipeline. Compared to the local execution, a new time-consuming phase named communication is introduced into the processing pipeline of the remote execution besides image conversion and inference phases. We obtain similar observations to (3) and (4). In addition, (16) Fig. 14(a) shows that only one core of the smartphone's processor reaches to the highest possible frequency under the conservative governor, which indicates that running CNN models on the edge server is capable of reducing the workload on the smartphone's CPUs. (17) However, because of the workload reduction and the conservative governor's low-frequency bias, the per frame latency of the remote execution under the conservative governor is approximately 10.3% larger than that of the local execution, as shown in Table III. This observation demonstrates that the

TABLE III
LATENCY RESULTS OF THE REMOTE EXECUTION WITH DIFFERENT CPU GOVERNORS

CPU Governor	Conservative	Ondemand	Interactive	Userspace	Powersave	Performance
Per Frame Latency (second)	1.009	0.853	0.905	1.075	4.214	**0.819**
Image Conversion Latency (second)	0.571	0.409	0.477	0.666	3.622	**0.376**
Inference Latency (Second)	0.189	0.189	**0.179**	0.180	0.185	0.192
Communication Latency (Second)	0.205	0.210	0.200	**0.174**	0.293	0.207
Others (Second)	0.044	0.045	0.049	0.055	0.114	**0.044**
Per Frame Latency Reduction (%)	-10.3	5.6	10.7	25.6	**45.7**	0.5

TABLE IV
SMARTPHONES AND THE EDGE SERVER USED IN THIS EXPERIMENT

Manufacturer	Samsung	Google	Asus	Nvidia
Model	Galaxy S5	Nexus 6	ZenFone AR	Jetson AGX Xavier
OS	Android 6.0.1	Android 5.1.1	Android 7.0	Ubuntu 18.04 LTS aarch64
SoC	Snapdragon 801 (28 nm)	Snapdragon 805 (28 nm)	Snapdragon 821 (14 nm)	Xavier
CPU	32-bit 4-core 2.5GHz Krait 400	32-bit 4-core 2.7GHz Krait 450	64-bit 4-core 2.4GHz Kryo	64-bit 8-core 2.26GHz Carmel
GPU	578MHz Adreno 330	600MHz Adreno 420	653MHz Adreno 530	512-core 1377MHz Volta with 64-TensorCores
RAM	2GB	3GB	6GB	16GB
WiFi	802.11n/ac, MIMO 2×2	802.11n/ac, MIMO 2×2	802.11n/ac/ad, MIMO 2×2	—
Release date	April 2014	November 2014	July 2017	September 2018

TABLE V
CLASSIFICATION & LATENCY RESULTS OF DIFFERENT SMARTPHONES

Smartphone		S5	Nexus 6	ZenFone AR
CPU Score		36871	37521	**58531**
GPU Score		6678	18063	**67286**
Image Processing Score		3103	6862	**11321**
Total Score		66414	80047	**173472**
Class		Low-end	Low-end	High-end
Per Frame Latency (s)	Local	1.098	1.013	**0.225**
	Remote	0.956	0.905	**0.312**
Latency Reduction (%)		**12.9**	10.7	-38.7

conservative governor is not suitable for the remote execution either and performs worse in the remote execution.

Interestingly, we find that (18) the remote execution achieves significantly distinct per frame latency reduction when the smartphone works on different CPU governors. For example, as shown in Table III, the remote execution achieves a per frame latency reduction of 45.7% in the powersave governor compared to the local execution, while it only obtains a per frame latency reduction of 0.5% in the performance governor. This observation may infer that *locally executing CNN-based object detection on the smartphone with advanced processors and working on a high CPU frequency is capable of achieving a comparable latency performance as the remote execution.* This inference is important for guiding whether a smartphone has to offload its object detection tasks to the edge server for reducing the service latency.

In order to verify this inference, we conduct a measurement study using three smartphones with different computation capacities, where their characteristics are summarized in Table IV. We classify them into two classes, *low-end* and *high-end* smartphones, according to their general hardware performance tested by using an Antutu benchmark [56]. The testing results are shown in Table V. All these three smartphones work on the interactive governor. The results verify our inference above, where the per frame latency of the low-end smartphones is decreased around 12%, whereas the per frame latency of the high-end smartphone is increased approximately 38.7% when offloading the object detection tasks to the edge server (note that the value of the latency reduction may differ depending on how powerful the edge server's GPU is). This observation supports the fact that lots of recently released smartphones with high computation power possess the capability of running a light CNN model with low latency. However, the detection accuracy of the large CNN model on the edge server is better than that of the light CNN model on the smartphone (e.g., $mAP = 51.5$ on the server and $mAP = 19.3$ on the smartphone when the frame resolution is around 300×300 pixels). Furthermore, in general, different use cases may have variant latency/accuracy requirements. For example, the AR cognitive assistance case where a high-end wearable device helps visually impaired people to navigate on a street may need a low latency but can tolerate a relatively high number of false positives (i.e., false alarms are fine but missing any potential threats on the street is costly) [37]. In contrast, an AR used for recommending products in shopping malls or supermarkets may tolerate a relatively long latency but require high detection accuracy. *Therefore, both the smartphone's computation capacity and the use case should be considered when determining the appropriate execution approach (i.e., local or remote).*

Per Frame Energy Consumption: We next explore how the CPU governor impacts the per frame energy consumption of object detection in the remote execution scenario, where the smartphone works on the interactive CPU governor. The experimental results are shown in Fig. 15, where Figs. 15(a)–15(f) depict the power consumption; Figs. 15(g)–15(l) illustrate the average per frame energy consumption; and Figs. 15(m)–15(r) show the average percentage breakdown of energy consumed by each phase in the processing pipeline. We find that (19) the remote execution decreases the power consumption compared to the local execution when the smartphone works on conservative, ondemand, interactive, and performance CPU governors. However, when the smartphone works on userspace

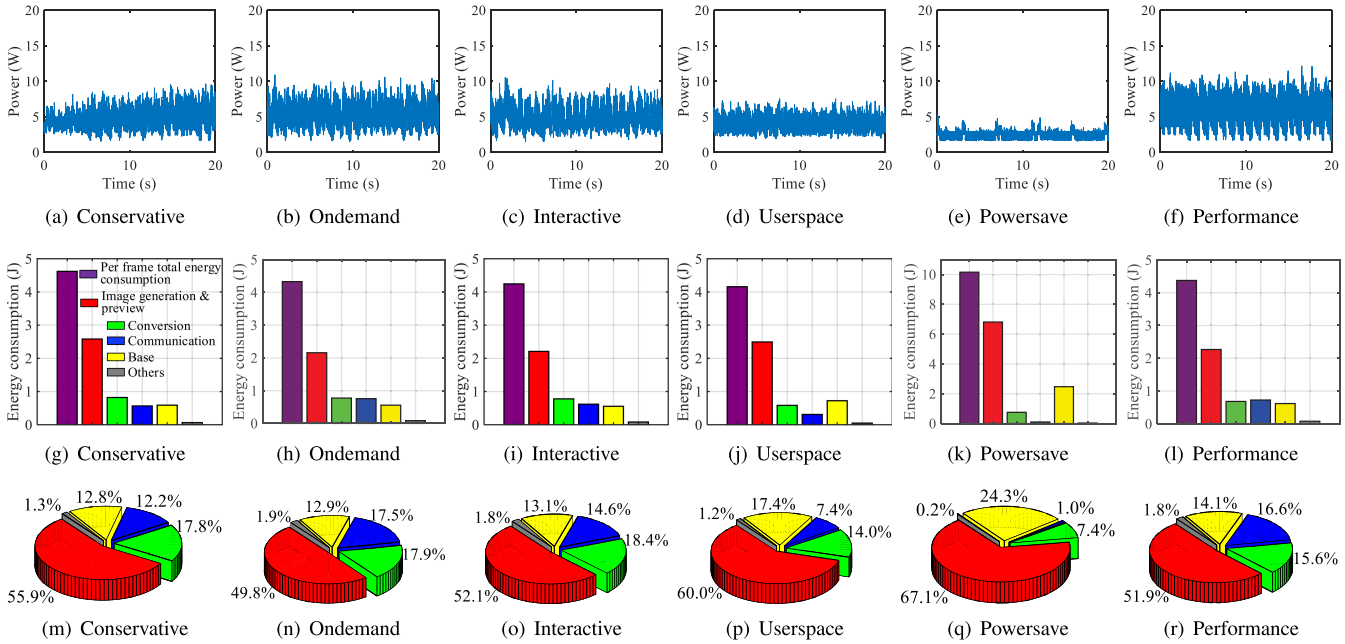


Fig. 15. CPU governor vs. power and average per frame energy consumption (CNN model size: 320 × 320 pixels).

TABLE VI
PER FRAME ENERGY CONSUMPTION RESULTS OF THE REMOTE EXECUTION WITH DIFFERENT CPU GOVERNORS

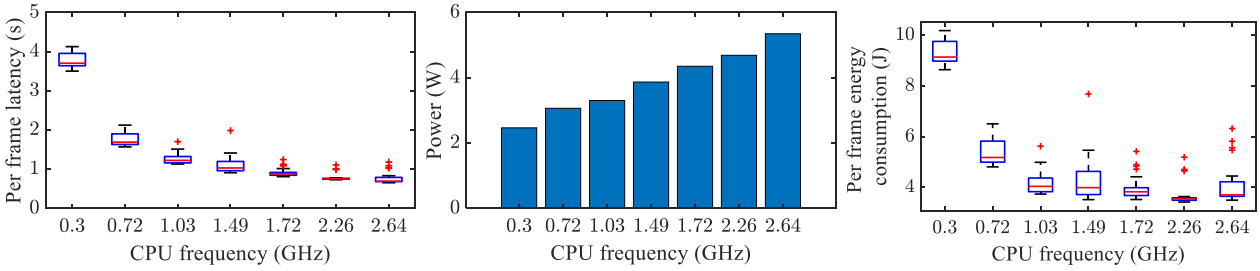
CPU Governor	Conservative	Ondemand	Interactive	Userspace	Powersave	Performance
Power Consumption (watt)	4.579	5.074	4.728	3.867	**2.410**	5.341
Per Frame Energy Consumption (Joule)	4.620	4.328	4.278	**4.157**	10.156	4.375
Power Consumption Reduction (%)	**14.5**	11.4	12.7	-1.4	-4.4	12.7
Per Frame Energy Consumption Reduction (%)	12.6	16.4	22.0	24.5	**43.4**	13.1

and powersave CPU governors, the remote execution consumes more power than the local execution, as shown in Table VI. This observation is a supplement to observation (16), which indicates that *(i) offloading the object detection tasks to the edge server may not be able to reduce the workload on the smartphone when the smartphone's CPUs run at a low frequency; (ii) the communication phase (i.e., remote execution) is more power-consuming than the inference phase (i.e., local execution) when the CPU frequency is low.* (20) As depicted in Figs. 15(g)-15(l) and Table VI, the remote execution is capable of reducing the per frame energy consumption compared to the local execution when the smartphone works on these six tested CPU governors. In addition, observation (18) and its corresponding inference are also applicable for the per frame energy consumption.

Interestingly, (21) the userspace governor (i.e., the CPU frequency is set to 1.49GHz) achieves the lowest per frame energy consumption in the remote execution, as illustrated in Figs. 15(g)–15(l) and Table VI. This observation is different from the local execution, where the CPU with the highest frequency achieves the lowest per frame energy consumption. We conduct an experiment study to explore the reason, where we set the test smartphone to the userspace governor and gradually raise its CPU frequency from the lowest to the highest. The experimental results are shown in Fig. 16. We find that (22) the higher the CPU frequency, the lower per frame latency the smartphone derives and the higher power it consumes. However, the reduction of the per frame latency and the increase of the power consumption are disproportional, as depicted in Figs. 16(a) and 16(b). For example, as compared to 2.26GHz, 2.64GHz only reduces about 5% latency but increases about 14% power consumption. As compared to 0.3GHz, 0.72GHz reduces about 55% latency but only increases about 24% power consumption. *This observation advocates adapting the smartphone's CPU frequency for the per frame latency reduction by trading as little increase of the per frame energy consumption as possible.* For example, Fig. 16(c) illustrates that selecting the CPU frequency around 2.26GHz achieves the lowest per frame energy consumption, a comparable per frame latency, and a lower power consumption compared to 2.64GHz.

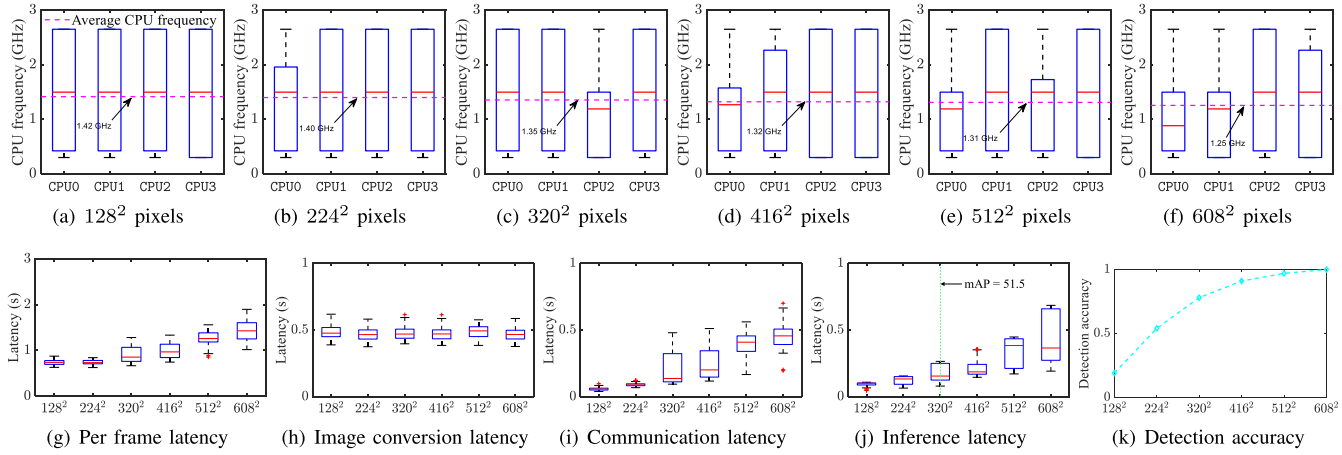
B. The Impact of CNN Model Size

Per Frame Latency: In this experiment, we implement six object detection models based on the YOLOv3 framework [3] with different CNN model sizes (i.e., from 128 × 128 to 608 × 608 pixels). The test smartphone works on the default CPU governor, interactive. Fig. 17 depicts the per frame latency of running CNN-based object detection with different model sizes in the remote execution. We make the following observations. (23) In contrary to the local execution, raising the CNN model size in the remote execution decreases the average CPU frequency, as shown in Figs. 17(a)–17(f). This is because



(a) CPU frequency vs. per frame latency (b) CPU frequency vs. power consumption (c) CPU frequency vs. per frame energy consumption

Fig. 16. Performance variations with increasing the CPU frequency in remote execution (CNN model size: 320 × 320 pixels).



(a) 128^2 pixels (b) 224^2 pixels (c) 320^2 pixels (d) 416^2 pixels (e) 512^2 pixels (f) 608^2 pixels

(g) Per frame latency (h) Image conversion latency (i) Communication latency (j) Inference latency (k) Detection accuracy

Fig. 17. CNN model size vs. CPU frequency, latency, and detection accuracy (CPU governor: interactive).

the smartphone experiences a relatively long idle period (i.e., waiting for the detection results from the edge server) when the CNN model size is large (i.e., a long inference latency at the edge server side). In WiFi networks, when transmitting a single image frame, the smartphone's wireless interface experiences four phases: promotion, data transmission, tail, and idle. When an image transmission request comes, the wireless interface enters the promotion phase. Then, it enters the data transmission phase to send the image frame to the edge server. After completing the transmission, the wireless interface is forced to stay in the tail phase for a fixed duration and waits for other data transmission requests and the detection results. If the smartphone does not receive the detection result in the tail phase, it enters the idle phase and waits for the feedback from its associated edge server. *Therefore, in contrary to the local execution, using a large CNN model size in the remote execution can extend the battery life and improve the detection accuracy.* (24) Similar to the local execution, a larger CNN model size always results in a higher per frame latency in the remote execution, where the per frame latency increment is mainly from the raise of the communication and the inference latency, as shown in Figs. 17(g)–17(j). In addition, Fig. 17(k) depicts the detection accuracy of the YOLO under different CNN model sizes, where the detection accuracy is defined as the ratio of the number of correctly recognized objects to that of the total objects in an image frame (on calculating the accuracy, we assume that the YOLO is capable of detecting all objects in an image frame when the CNN model size is 608 × 608 pixels). We find that (25) although a higher CNN model size enables a better detection accuracy, the accuracy gain narrows down at a high CNN model size. However, (26) the speed of the per frame latency and the inference latency increases becomes faster at a higher CNN model size, as illustrated in Figs. 17(g) and 17(j). *These two observations inspire us to trade detection accuracy (i.e., mAP) for the per frame latency reduction when the CNN model size is large.*

Per Frame Energy Consumption: We next investigate how the CNN model size impacts the per frame energy consumption in the remote execution. Fig. 18 shows the measured energy consumption results, where the smartphone works on the interactive CPU governor. We observe that (27) the remote execution saves approximately 52.5% per frame energy on average when the frame resolution is 608 × 608 pixels, as shown in Table VII. However, it consumes slightly more per frame energy than the local execution when the frame resolution is 128 × 128 pixels. This observation is rather significant, which demonstrates that *running CNN-based object detection remotely does not always consume less energy than the local execution.* In addition, (28) the larger the CNN model size, the more per frame energy reduction the remote execution derives compared to the local execution. For example, running a 224 × 224 pixels model only reduces about 14.1% per frame energy, while executing a 608 × 608 pixels model decreases about 52.5% per frame energy consumption. Therefore, in

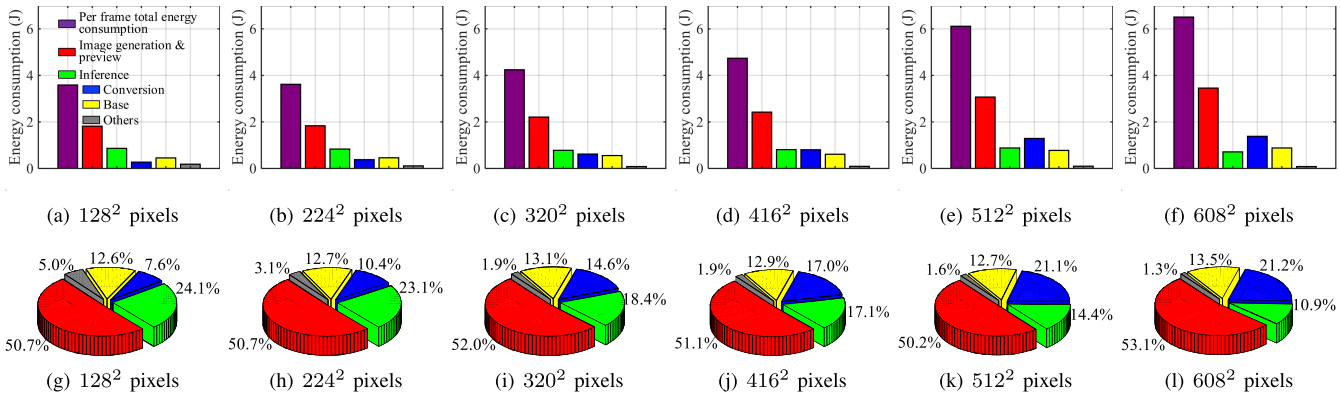
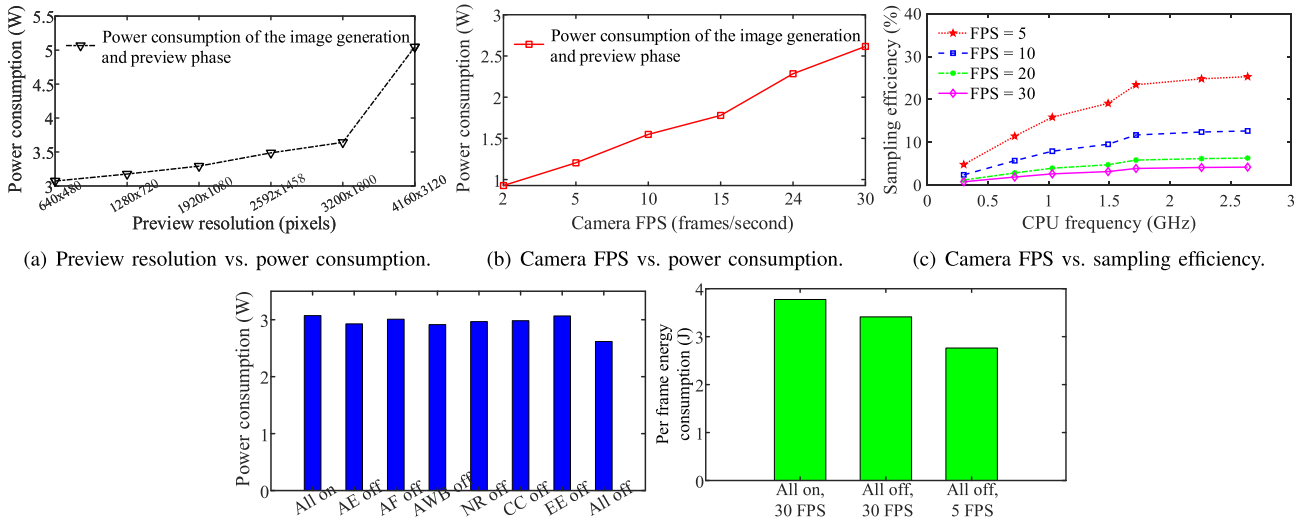


Fig. 18. CNN model size vs. per frame energy consumption (CPU governor: interactive).

TABLE VII
PER FRAME ENERGY CONSUMPTION RESULTS OF THE REMOTE EXECUTION WITH DIFFERENT CNN MODEL SIZES

CNN Model Size (pixels)		128 × 128	224 × 224	320 × 320	416 × 416	512 × 512	608 × 608
Per Frame Energy Consumption (J)	Local	**3.584**	4.210	5.923	7.961	11.169	13.699
	Remote	**3.586**	3.616	4.242	4.736	6.111	6.508
Per Frame Energy Consumption Reduction (%)		0	14.1	28.4	40.5	45.3	**52.5**



(a) Preview resolution vs. power consumption. (b) Camera FPS vs. power consumption. (c) Camera FPS vs. sampling efficiency.

(d) 3A and image post processing algorithms vs. power consumption. (e) Comparison of the per frame energy consumption.

Fig. 19. Power consumption analyses of image generation and preview phases (remote execution).

order to take the best advantage of the remote execution, executing a CNN with a larger model size is recommended.

C. The Impact of Image Generation and Preview

RQ 3: Besides the network condition, what else impacts the energy consumption and latency when executed remotely, and how? As we presented in the aforementioned observations, the image generation and preview is the most energy-consuming phase in both local and remote execution scenarios. Thus, to improve the energy efficiency of the object detection processing pipeline, we must reduce the energy consumption of image generation and preview phases. We seek to understand the interactions between the power consumption and various factors (e.g., the preview resolution, 3A, and several image post processing algorithms) as follows.

Preview Resolution vs. Power Consumption: We first examine how the preview resolution influences the power consumption of image generation and preview phases, as shown in Fig. 19(a). We find that (29) as the preview resolution grows, the power consumption increases dramatically. Therefore, a preview with a higher frame resolution on the smartphone provides a better quality preview for users, but at the expense of battery drain, which is applicable for both local and remote execution cases.

Camera FPS vs. Power Consumption: We next vary the smartphone's camera FPS to explore how it impacts the device's power consumption, where the camera FPS is defined as the number of frames that the camera samples per second. Fig. 19(b) shows that (30) a large camera FPS leads to a high power consumption. However, as shown in Fig. 2, not every camera captured image frame is sent to the edge

TABLE VIII
IMAGE CONVERSION LATENCY RESULTS WITH DIFFERENT CONVERSION METHODS

Preview Resolution (pixels)		320 × 240	352 × 288	640 × 480	720 × 480	800 × 480	800 × 600	1024 × 768
Conversion Latency (ms)	Java	130.1	175.9	484.9	539.0	596.6	711.9	1157.1
	C	14.7	16.4	40.3	48.3	48.4	54.4	82.6
Latency Reduction (%)		88.7	90.7	91.7	91.0	91.9	92.4	92.9

(a) All enabled. (b) All disabled.

Fig. 20. Comparison of the object detection results (remote execution).

server for detection. Because of the need (i) to avoid the processing of stale frames and (ii) to decrease the transmission energy consumption, only the latest camera sampled image frame is transmitted to the server. This may result in the smartphone expending significant reactive power for sampling non-detectable image frames. In Fig. 19(c), we quantify the sampling efficiency with the variation of the camera FPS. As we expected, (31) a large camera FPS leads to a lower sampling efficiency (e.g., less than 2% of the power is consumed for sampling the detectable image frames when the camera FPS is set to 30). However, in most mobile AR applications, users usually request a high camera FPS for a smoother preview experience, which is critical for tracking targets in physical environments. Interestingly, (32) increasing CPU frequency can reduce the reactive power for sampling, as shown in Fig. 19(c). These observations demonstrate that when a high camera FPS is requested, increasing CPU frequency can promote the sampling efficiency but may also boost the power consumption. Therefore, finding a CPU frequency that can balance this tradeoff is critical.

Image Post Processing and 3A Algorithms vs. Power Consumption: Lastly, we examine the effect of multiple image post processing and 3A algorithms on the power consumption of image generation and preview phases, as shown in Figs. 19(d) and 19(e). Note that when the AE is disabled, we manually set the camera ISO and exposure time to 400 and 20 ms, respectively. We observe that (33) disabling the 3A, NR, CC, and EE algorithms decreases the power consumption by 14.8%. We conduct another experiment to understand if disabling these algorithms would impact the object detection performance. As shown in Fig. 20, (34) the detection performance does not degrade. Furthermore, we compare the per frame energy consumption among three cases, as depicted in Fig. 19(e): 1) all enabled with camera capture frame rate 30; 2) all disabled with camera capture frame rate 30; and 3) all disabled with camera capture frame rate 5. We find that (35) the per frame energy consumption of the second and the third cases decreases by approximately 10% and 27%, respectively, compared to the first case. *Therefore, these three observations may answer the question that we presented in Section IV-A: these energy-hungry image post processing algorithms may not be necessary for camera captured image frames to achieve successful object detection results.*

D. The Impact of the Image Conversion Method

As depicted in Tables I and III, the image conversion phase is one of the most time-consuming phases in both local and remote execution scenarios. This is because the image conversion method that we implemented in the testbed is developed based on Java, which is inefficient and slow. Thus, in order to improve the efficiency of the image conversion, we implement image conversion based on C in Android Native Development Kit (NDK). We compare these two methods by measuring their conversion latency with different preview resolutions. The measurement results are presented in Table VIII. We find that the image conversion method developed based on C decreases the image conversion latency by over 90%.

E. Insights and Research Opportunities

Insights:

- Offloading the object detection to the edge server does not always reduce the per frame latency and energy consumption of the mobile AR client compared to the local execution. For example, as we observed in our experiments, locally running a detection model with a size of 100 × 100 pixels achieves lower per frame latency and energy consumption than the remote execution that runs a CNN with a similar model size. In addition, the specific CNN model size when the local execution has better performance than the remote execution may vary with the computation capacities of the edge server and mobile AR clients, and even the wireless network bandwidth.
- In the remote execution, the mobile AR client does not achieve the lowest per frame energy consumption when its CPU is set to the highest frequency, which is different from the local execution. For example, in our experiment, the lowest per frame energy consumption is obtained when the CPU frequency is around 2.26GHz. Although this value may be different for diverse smartphones or wearable AR devices, this knowledge is important for designing the CPU scaling mechanism.

Research Opportunities:

- The energy consumption of the communication phase becomes the second largest portion of the per frame energy consumption when the frame resolution of the offloaded image is large (determined by the CNN model

size). Thus, improving the image transmission energy efficiency is a potential research issue for the remote execution. For example, as we presented, when transmitting an image frame, the mobile AR client's wireless interface experiences four phases: 1) promotion; 2) data transmission; 3) tail; and 4) idle. After completing the transmission, the wireless interface is forced to stay in the tail phase for a fixed duration and waits for other data transmission requests and the detection results. Therefore, developing a mechanism that can adaptively adjust the duration of the tail phase based on the predicted inference latency at the edge server and background activities of the mobile AR client may possibly improve the energy efficiency of the mobile AR client by allowing it to enter the idle phase faster.

- Although our experimental results indicate that some energy-consuming image post processing algorithms may not be necessary for mobile AR clients to achieve successful object detection results, more comprehensive studies are required to investigate this issue. For example, is this result influenced by other factors, such as the object category, frame resolution, and object detection algorithm?

VI. Threats to Validity

External validity: External validity can be criticized by using a single version of Android OS on the tested Google Nexus 6 smartphone. The threat is mitigated by running our benchmark applications on multiple smartphones with significantly different computation capacity (i.e., high-end and low-end), as described in Table IV. Furthermore, although the numerical values of our measurement with a specific experiment configuration (e.g., the per frame energy consumption of locally executing a 320×320 CNN model with Interactive CPU governor) cannot be generalized to all possible smartphones, such as iPhone 12 and Samsung Galaxy Note20, this article focuses on investigating the trend of how the smartphone's energy consumption may vary when our benchmark applications are executed with different configurations, which will help predict variations on the energy consumption of other smartphones. In addition, an architecture-level comparison is not conducted in this article (e.g., comparing the energy efficiency of running the benchmark applications on an iPhone with an Apple's bionic chip and an Android phone with a Samsung's Exynos chip). Because comparing different architectures is more challenging and needs more efforts on the hardware setup (e.g., selecting appropriate smartphones and using different ways to connect the power supply to each smartphone based on their different circuit designs) as well as the experiment design, we leave it to our future work. External validity may also be threatened by using a custom-designed object detection benchmark instead of real-world applications. However, our benchmark applications exercise most of the main functionalities of existing and potential mobile object detection applications, such as image generation, camera preview, image conversion, inference, data transmission, and virtual content rendering, which means that our custom-designed object detection benchmark applications are representative.

Internal validity: The collected power consumption and CPU frequency data might be influenced by the background activities. We mitigate this threat by terminating all other optional applications and services that can impact the smartphone's workload. One of the main contributions of this article is comparing the energy consumption and latency of executing CNN-based object detections locally and remotely. To have a fair comparison between the local and remote executions, each comparison is conducted with the same configurations (e.g., CPU governor, CNN model size, preview resolution, and camera sampling rate) and under the same conditions. For instance, all the power measurements are conducted in a constant temperature laboratory. In addition, the temperature of the tested smartphone's CPU may increase when running the benchmark application, which may impact the power consumption of the smartphone. To mitigate this threat, a new measurement is launched only if the temperature of the CPU cools down to around 42°C after the previous measurement.

Construct validity: Dissecting the energy consumption for each phase in an application is difficult. The accuracy of our evaluation is guaranteed by the energy measurement strategy we proposed in Section III-C, including the local clock synchronization and cooling-off period. Specifically, the precision of the local clock synchronization is in millisecond. In addition, we hypothesize that the power consumption is influenced by the workload accumulation, which is the assumption for breaking down the power consumption of the three parallel executions in our benchmark applications.

VII. Conclusion

In this article, we presented the first detailed experimental study of the energy consumption and the performance of a CNN-based object detection application. We examined both local and remote execution cases. We found that the performance of object detection is heavily affected by various factors, such as CPU governor, CPU frequency, and CNN model size. Although executing object detection on remote edge servers is one of the most commonly used approaches to assist low-end smartphones in improving their energy efficiency and performance, contrary to our expectation, local execution may consume less energy and obtain lower latency, as compared to remote execution. Overall, we believe that our findings provide great insights and guidelines to the future design of energy-efficient processing pipeline of CNN-based object detection.

References

[1] L. N. Huynh, Y. Lee, and R. K. Balan, "Deepmon: Mobile GPU-based deep learning framework for continuous vision applications," in *Proc. ACM Mobisys*, 2017, pp. 82–95.

[2] D. Chatzopoulos, C. Bermejo, Z. Huang, and P. Hui, "Mobile augmented reality survey: From where we are to where we go," *IEEE Access*, vol. 5, pp. 6917–6950, 2017.

[3] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," 2018. [Online]. Available: arXiv:1804.02767.

[4] *Tensorflow Lite*. Accessed: Oct. 10, 2020. [Online]. Available: https://www.tensorflow.org/lite/

[5] T. Y.-H. Chen, L. Ravindranath, S. Deng, P. Bahl, and H. Balakrishnan, "Glimpse: Continuous, real-time object recognition on mobile devices," in *Proc. ACM Sensys*, 2015, pp. 155–168.

[6] P. Jain, J. Manweiler, and R. R. Choudhury, "Low bandwidth offload for mobile AR," in *Proc. ACM CoNEXT*, 2016, pp. 237–251.

[7] H. Wang *et al.*, "Architectural design alternatives based on cloud/edge/fog computing for connected vehicles," *IEEE Commun. Surveys Tuts.*, vol. 22, no. 4, pp. 2349–2377, 4th Quart., 2020.

[8] A. Carroll and G. Heiser, "An analysis of power consumption in a smartphone," in *Proc. USENIX Annu. Tech. Conf.*, 2010, p. 21.

[9] S. K. Saha, P. Deshpande, P. P. Inamdar, R. K. Sheshadri, and D. Koutsonikolas, "Power-throughput tradeoffs of 802.11n/ac in smartphones," in *Proc. IEEE INFOCOM*, 2015, pp. 100–108.

[10] L. Liu, H. Li, and M. Gruteser, "Edge assisted real-time object detection for mobile augmented reality," in *Proc. ACM 25th Annu. Int. Conf. Mobile Comput. Netw. (MobiCom)*, 2019, pp. 1–16.

[11] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app? Fine grained energy accounting on smartphones with Eprof," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 29–42.

[12] W. Oliveira, R. Oliveira, F. Castor, B. Fernandes, and G. Pinto, "Recommending energy-efficient Java collections," in *Proc. IEEE/ACM 16th Int. Conf. Min. Softw. Repositories (MSR)*, 2019, pp. 160–170.

[13] S. Chowdhury, S. Borle, S. Romansky, and A. Hindle, "GreenScaler: Training software energy models with automatic test generation," *Empir. Softw. Eng.*, vol. 24, no. 4, pp. 1649–1692, 2019.

[14] A. Hindle, A. Wilson, K. Rasmussen, E. J. Barlow, J. C. Campbell, and S. Romansky, "Greenminer: A hardware based mining software repositories software energy consumption framework," in *Proc. ACM 11th Work. Conf. Min. Softw. Repositories*, 2014, pp. 12–21.

[15] T. Agolli, L. Pollock, and J. Clause, "Investigating decreasing energy usage in mobile apps via indistinguishable color changes," in *Proc. IEEE/ACM 4th Int. Conf. Mobile Softw. Eng. Syst. (MOBILESoft)*, 2017, pp. 30–34.

[16] M. Wan, Y. Jin, D. Li, and W. G. J. Halfond, "Detecting display energy hotspots in android apps," in *Proc. IEEE 8th Int. Conf. Softw. Test. Verification Validation (ICST)*, 2015, pp. 1–10.

[17] D. Li, Y. Lyu, J. Gui, and W. G. J. Halfond, "Automated energy optimization of HTTP requests for mobile applications," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. (ICSE)*, 2016, pp. 249–260.

[18] S. A. Chowdhury, V. Sapra, and A. Hindle, "Client-side energy efficiency of HTTP/2 for Web and mobile app developers," in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reeng. (SANER)*, vol. 1, 2016, pp. 529–540.

[19] A. McIntosh, S. Hassan, and A. Hindle, "What can android mobile app developers do about the energy consumption of machine learning?" *Empir. Softw. Eng.*, vol. 24, no. 2, pp. 562–601, 2019.

[20] Y. Xiao *et al.*, "Modeling energy consumption of data transmission over Wi-Fi," *IEEE Trans. Mobile Comput.*, vol. 13, no. 8, pp. 1760–1773, Aug. 2013.

[21] H. Wang, J. Xie, and X. Liu, "Rethinking mobile devices' energy efficiency in WLAN management services," in *Proc. IEEE SECON*, 2018, pp. 370–378.

[22] J. Huang, F. Qian, A. Gerber, Z. M. Mao, S. Sen, and O. Spatscheck, "A close examination of performance and power characteristics of 4G LTE networks," in *Proc. ACM MobiSys*, 2012, pp. 225–238.

[23] H. Wang, J. Xie, and T. Han, "A smart service rebuilding scheme across cloudlets via mobile AR frame feature mapping," in *Proc. IEEE ICC*, 2018, pp. 1–6.

[24] H. Wang, J. Xie, and T. Han, "V-handoff: A practical energy efficient handoff for 802.11 infrastructure networks," in *Proc. IEEE ICC*, 2017, pp. 1–6.

[25] A. Shye, B. Scholbrock, and G. Memik, "Into the wild: Studying real user activity patterns to guide power optimizations for mobile architectures," in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2009, pp. 168–178.

[26] M. J. Walker *et al.*, "Accurate and stable run-time power modeling for mobile and embedded CPUs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 36, no. 1, pp. 106–119, Jan. 2017.

[27] K. DeVogeleer, G. Memmi, P. Jouvelot, and F. Coelho, "Modeling the temperature bias of power consumption for nanometer-scale CPUs in application processors," in *Proc. IEEE Int. Conf. Embedded Comput. Syst. Archit. Model. Simulati. (SAMOS XIV)*, 2014, pp. 172–180.

[28] F. Xu, Y. Liu, Q. Li, and Y. Zhang, "V-edge: Fast self-constructive power modeling of smartphones based on battery voltage dynamics," in *Proc. USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, 2013, pp. 43–55.

[29] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang, "Fine-grained power modeling for smartphones using system call tracing," in *Proc. 6th Conf. Comput. Syst.*, 2011, pp. 153–168.

[30] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 91–99.

[31] R. J. Wang, X. Li, and C. X. Ling, "Pelee: A real-time object detection system on mobile devices," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 1963–1972.

[32] X. Zhang, X. Zhou, M. Lin, and J. Sun, "ShuffleNet: An extremely efficient convolutional neural network for mobile devices," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, 2018, pp. 6848–6856.

[33] D. Chen *et al.*, "Federated learning based mobile edge computing for augmented reality applications," in *Proc. IEEE ICNC*, 2020, pp. 767–773.

[34] J. Huang *et al.*, "Speed/accuracy trade-offs for modern convolutional object detectors," in *Proc. IEEE CVPR*, 2017, pp. 3296–3297.

[35] Q. Liu, S. Huang, J. Opadere, and T. Han, "An edge network orchestrator for mobile augmented reality," in *Proc. IEEE INFOCOM*, 2018, pp. 756–764.

[36] A. G. Howard *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017. [Online]. Available: arXiv:1704.04861.

[37] X. Ran, H. Chen, Z. Liu, and J. Chen, "Delivering deep learning to mobile devices via offloading," in *Proc. ACM Workshop Virtual Reality Augmented Reality Netw.*, 2017, pp. 42–47.

[38] W. Hu and G. Cao, "Energy-aware CPU frequency scaling for mobile video streaming," in *Proc. IEEE ICDCS*, 2017, pp. 2314–2321.

[39] W. Hu and G. Cao, "Energy optimization through traffic aggregation in wireless networks," in *Proc. IEEE INFOCOM*, 2014, pp. 916–924.

[40] N. C. Luong, P. Wang, D. Niyato, Y. Wen, and Z. Han, "Resource management in cloud networking using economic analysis and pricing models: A survey," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 2, pp. 954–1001, 2nd Quart., 2017.

[41] H. Wang, B. Kim, J. Xie, and Z. Han, "E-auto: A communication scheme for connected vehicles with edge-assisted autonomous driving," in *Proc. IEEE ICC*, 2019, pp. 1–6.

[42] Y. Geng, Y. Yang, and G. Cao, "Energy-efficient computation offloading for multicore-based mobile devices," in *Proc. IEEE INFOCOM*, 2018, pp. 46–54.

[43] A. P. Miettinen and J. K. Nurminen, "Energy efficiency of mobile clients in cloud computing," in *Proc. HotCloud*, vol. 10, 2010, p. 4.

[44] X. Ran, H. Chen, X. Zhu, Z. Liu, and J. Chen, "Deepdecision: A mobile deep learning framework for edge video analytics," in *Proc. IEEE INFOCOM*, 2018, pp. 1421–1429.

[45] J. Hanhirova, T. Kämäräinen, S. Seppälä, M. Siekkinen, V. Hirvisalo, and A. Ylä-Jääski, "Latency and throughput characterization of convolutional neural networks for mobile computer vision," in *Proc. 9th ACM Multimedia Syst. Conf.*, 2018, pp. 204–215.

[46] J.-J. Chen, C.-Y. Yang, T.-W. Kuo, and C.-S. Shih, "Energy-efficient real-time task scheduling in multiprocessor DVS systems," in *Proc. IEEE Asia South Pac. Design Autom. Conf.*, 2007, pp. 342–349.

[47] J. Kwak, O. Choi, S. Chong, and P. Mohapatra, "Dynamic speed scaling for energy minimization in delay-tolerant smartphone applications," in *Proc. IEEE INFOCOM*, 2014, pp. 2292–2300.

[48] W. Y. Lee, "Energy-saving DVFS scheduling of multiple periodic real-time tasks on multi-core processors," in *Proc. 13th IEEE/ACM Int. Symp. Distrib. Simulat. Real Time Appl.*, 2009, pp. 216–223.

[49] J. Redmon. (2013–2016). *Darknet: Open Source Neural Networks in C*. [Online]. Available: http://pjreddie.com/darknet/

[50] T.-Y. Lin *et al.*, "Microsoft COCO: Common objects in context," in *Proc. Eur. Conf. Comput. Vis.*, 2014, pp. 740–755.

[51] *Monsoon Power Monitor*. Accessed: Oct. 10, 2020. [Online]. Available: https://www.msoon.com/

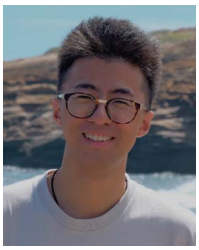
[52] Android. *ImageFormat*. Accessed: Oct. 2020. [Online]. Available: https://developer.android.com/reference/android/graphics/ImageFormat

[53] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes (VOC) challenge," *Int. J. Comput. Vis.*, vol. 88, no. 2, pp. 303–338, 2010.

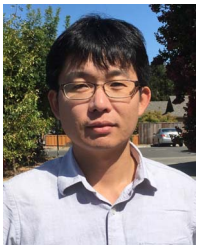
[54] H. Wang and J. Xie, "User preference based energy-aware mobile AR system with edge computing," in *Proc. IEEE INFOCOM*, 2020, pp. 1–10.

[55] H. Wang, B. Kim, J. Xie, and Z. Han, "How is energy consumed in smartphone deep learning apps? Executing locally vs. remotely," in *Proc. IEEE Globecom*, 2019, pp. 1–6.

[56] *Antutu Benchmark*. Accessed: Oct. 10, 2020. [Online]. Available: https://www.antutu.com/en/



Haoxin Wang received the B.S. degree in control science and engineering from the Harbin Institute of Technology, Harbin, China. He is currently pursuing the Ph.D. degree with the Department of Electrical and Computer Engineering, University of North Carolina at Charlotte. His research interests include mobility management of protocol design, modeling, system prototyping of mobile-edge computing networks, mobile augmented reality, and connected vehicles.



BaekGyu Kim (Member, IEEE) received the B.S. and M.S. degrees in electrical engineering and computer science from Kyungpook National University, South Korea, and the Ph.D. degree in computer science from the University of Pennsylvania. He is currently a Principal Researcher with Toyota Motor North America, InfoTech Labs. His research area includes software platform technologies for connected cars, and model-based software development for high-assurance systems.

Jiang Xie (Fellow, IEEE) received the B.E. degree in electrical and computer engineering from Tsinghua University, Beijing, China, the M.Phil. degree in electrical and computer engineering from the Hong Kong University of Science and Technology, and the M.S. and Ph.D. degrees in electrical and computer engineering from the Georgia Institute of Technology. She joined the Department of Electrical and Computer Engineering, University of North Carolina at Charlotte (UNC-Charlotte) as an Assistant Professor in August 2004, where she is currently a Full Professor. Her current research interests include resource and mobility management in wireless networks, mobile computing, Internet of Things, and cloud/edge computing. She received the U.S. National Science Foundation NSF Faculty Early Career Development (CAREER) Award in 2010, the Best Paper Award from IEEE Global Communications Conference in 2017, the Best Paper Award from IEEE/WIC/ACM International Conference on Intelligent Agent Technology in 2010, and the Graduate Teaching Excellence Award from the College of Engineering at UNC-Charlotte in 2007. She is on the editorial boards of the IEEE/ACM TRANSACTIONS ON NETWORKING and *Journal of Network and Computer Applications* (Elsevier). She is a Senior Member of ACM.

Zhu Han (Fellow, IEEE) received the B.S. degree in electronic engineering from Tsinghua University, in 1997, and the M.S. and Ph.D. degrees in electrical and computer engineering from the University of Maryland at College Park in 1999 and 2003, respectively.

From 2000 to 2002, he was a Research and Development Engineer with JDSU, Germantown, MD, USA. From 2003 to 2006, he was a Research Associate with the University of Maryland at College Park. From 2006 to 2008, he was an Assistant Professor with Boise State University, Boise, ID, USA. He is currently a John and Rebecca Moores Professor with the Electrical and Computer Engineering Department as well as with the Computer Science Department, University of Houston, Houston, TX, USA. His research interests include wireless resource allocation and management, wireless communications and networking, game theory, big data analysis, security, and smart grid. He received the NSF Career Award in 2010, the Fred W. Ellersick Prize of the IEEE Communication Society in 2011, the EURASIP Best Paper Award for the *Journal on Advances in Signal Processing* in 2015, the IEEE Leonard G. Abraham Prize in the field of Communications Systems (Best Paper Award in IEEE JSAC) in 2016, and several best paper awards in IEEE conferences. He was also the Winner of the 2021 IEEE Kiyo Tomiyasu Award, for outstanding early to mid-career contributions to technologies holding the promise of innovative applications, with the following citation: "for contributions to game theory and distributed management of autonomous communication networks." He has been 1% Highly Cited Researcher according to Web of Science since 2017. He was an IEEE Communications Society Distinguished Lecturer from 2015 to 2018, an AAAS Fellow since 2019, and an ACM Distinguished Member since 2019.