AN EXPERIMENTAL STUDY ON OPTIMIZING ENERGY CONSUMPTION OF EDGE IOT DEVICES

by

Mujahid Khan, B.S.

A thesis submitted to the Graduate College of
Texas State University in partial fulfillment
of the requirements for the degree of
Master of Science
with a Major in Computer Science
December 2021

Committee Members:

Anne Hee Hiong Ngu, Chair

Byron Gao

Qijun Gu

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

DEDICATION

I dedicate this work to my loving parents and my supportive friends. A very special thank you to my mother Shagufta Khizar, my father Khizar Hayat Khan and my elder brother Muneeb Hayat Khan. Their exceptional support and overwhelming love has nothing but been a blessing to me throughout my time here away from home. I would also like to mention my friends Pseudo, GM, Paitha and Moi for making this journey of graduate studies on a foreign soil considerably easier to progress. I am very thankful to have these people in my life and I owe a significant portion of my success to them.

ACKNOWLEDGEMENTS

Firstly, I would like to express my gratitude towards my supervisor Dr. Anne H. Ngu, for her guidance and expertise over the course of this thesis. I would also like to thank all the faculty and staff members of the Department of Computer Science for providing an excellent research and learning environment.

I would like to express my appreciation for my friends and colleagues, especially Dr. Junye Wen, for being an amazing friend and helping me transition to life in the US. Lastly, I am extremely thankful for my family for being an immense pillar of support throughout my time here, even if they were thousands of miles away.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure **Page**

ABSTRACT

Recent advances in the world of IoT have significantly improved what one can do with the help of personal gadgets such as smartwatches and other edge devices. This improvement has lead to the inclusion of additional functionality like fall detection, temperature and motion sensing, etc. using an edge device. This in itself is not a bad thing, however, this does pose the problem of increased energy consumption on devices which are already constrained by limited battery. A portion of this energy is consumed during client-server communication which is an essential part of applications that either perform major computations from sensor data on the cloud or use the server to backup user data. We present an experimental study which discusses the possibility of optimizing energy consumption for edge IoT devices by reducing communication cost of applications by determining an optimal chunk size. We present a metric **energy-per-byte** or **EPB** which helps determine the optimal chunk size for respective edge devices. We also discuss a case study to (1) determine whether containerized applications are energy efficient and (2) if a coordination language framework like Lingua Franca can help design energy efficient applications. We evaluate our approach with the help of two applications running on different edge devices and softwares, and we show that we achieve considerable energy optimization depending on the application functionality and the hardware involved.

I. INTRODUCTION

In the past couple of years, IoT has made significant progress. From personal devices such as smartwatches and environments like smart homes, to industrially scaling projects like smart cities and smart autonomous cars. There has been a variety of applications of IoT edge devices to an extent that today, a considerably significant amount of people possess at least one such device —be it a smart car or a smartwatch that many regularly or routinely use.

Recent advances in the world of IoT have increased the amount of use cases that these personal devices such as smartwatches have. But having additional functionality also poses the problem of increased energy consumption on devices already constrained by limited battery. For example, a device logging sensor data, say temperature, in real-time is bound to use more energy if it starts logging data from a motion sensor as well.

This work studies the possibility of optimizing energy consumption for edge IoT devices by aiding the programmer in setting parameters that would ultimately lead to lesser energy consumption. We aim to focus on optimizing the communication cost of applications by determining the optimal chunk size, with which the data should be transferred. It also involves a case study comparing energy consumption of native app and the same app running in a container environment such as Docker [7].

There has been an extensive amount of work on energy optimization —both without and in an IoT setting. Xiao et al. [8] compared 3G and Wi-fi while Balasubramanian et al. [9] compared GSM, 3G and Wi-Fi with respect to energy consumption. However, neither of them compared energy consumption of respective data transfer technologies for throughput efficiency for the same task.

Gupta et al. [10] made a measurement study of energy consumption when using VoIP applications with Wi-Fi connection in smartphones and showed that power

saving mode in Wi-Fi together with intelligent scanning techniques can reduce energy consumption. Xiao et al. [8] measured energy consumption for video streaming on mobile apps and concluded that Wi-Fi is more efficient than 3G. There are other measurement studies as well which compare different modes of communication but none of them discuss energy consumption differences for different packet sizes.

The work that seems the most related to ours is by Friedman et al. [11] where they measured power and throughput performance of Bluetooth and Wi-Fi usage in smartphones. They concluded that power consumption is generally linear with the obtained throughput, and Bluetooth uses less energy than Wi-Fi. However, this study also showed that different hardware and different software have different results and there is no generic trend. They also concluded that an upper bound does exist, bottlenecked by the receiver, after which the sender expends more power retransmitting packets. This dependency of energy consumption on software is also discussed by Flinn and Satyanarayanan [10] who point out that "*There is growing consensus that advances in battery technology and low-power circuit design cannot, by themselves, meet the energy needs of future mobile computers* "[10]. This observation has been confirmed by recent advances in green software engineering, which demonstrated how the source of energy leaks can be software-related as well [12, 13, 14].

Our work, on the other hand, considers multiple constraints in a real-world IoT setting. Firstly, the transmission is not always continuous. The data may be communicated in regular or irregular intervals. Secondly, the amount of data to send depends on the amount of data collected by the sensors which may vary based on different applications. Thirdly, since the transmission may not be continuous, if the app demands it, based off energy consumption data from different chunk sizes, an optimal communication interval can be set to reduce the amount of buffering

required (if needed). Lastly, unique apps on different devices with different operating systems will have their own optimal throughput efficiency for energy consumption. Given the hardware configuration of such a device, our tool can determine the optimal chunk size for data transfer with respect to energy consumption.

We also present a case study on energy consumption of an application when it is run natively compared to it being run in a container environment. We also discuss the possible use and study of Lingua Franca, and argue which one of them is better in which circumstance.

Contributions

This thesis has the following contributions. It:

- proposes a method to estimate energy consumption of edge IoT devices,
- proposes a new approach to reduce energy consumption with the help of optimal chunk size,
- presents a new metric called Energy-Per-Byte or EPB,
- discusses a case study to compare energy consumption between native, Lingua Franca and container version of the same app,
- evaluates the effectiveness of this approach on two real-world applications of different domains.

The organization of the rest of the thesis as as follows. In chapter II, we will provide background information about edge IoT devices, energy usage and optimization, containers and middleware accessor frameworks. In chapter III, we briefly summarize the related work and chapter IV will discuss our methodology in detail. In chapter V, we will discuss our evaluation and results, and provide our reasoning. Chapter VI will discuss the case study of energy cost comparison between

native, Lingua Franca and Docker version of the same application. Finally in chapter VII, we will conclude our thesis and discuss possible future work.

II. LITERATURE REVIEW

Over the past few years, Internet of Things or IoT, has had a significant impact in our lives. It plays an important role whether it is a smartphone or a smarthome environment or just something as handy as a smartwatch. Different types of data are collected and exchanged among interconnected sensors/devices through modern communication network infrastructure connected by million of IoT nodes [1, 15, 16, 17, 18]. This direction of computing has already overtaken the traditional methods based on stationary computing [19]. As a paradigm, IoT expresses that most physical devices, such as smart phones, smart watches and other embedded devices are interconnected with each other. These devices communicate with data centers and exchange information —all while adhering to their routine tasks [16].

Following various popular technologies these days, such as smart homes, smart grid and smart healthcare, IoT has become one of the essential components of people's home and workplace existence. It will continue to impact the daily life of people and it's not just limited to technology. IoT has been reported to be one of the most important technologies that will impact US interests in 2025 [16]. The number of interconnected physical devices has already transcended the human population for a couple of years now. In 2012, there were 9 billion interconnected physical devices [19] and by the end of 2020, total number of interconnected devices rose up to 21.6 billion [20]. This rapid increase in the number of mobile devices suggested that conventional centralized cloud computing would struggle to satisfy the Quality of Service (QoS) for many applications. However, with the introduction of 5G technology, edge computing becomes a viable and key solution to solve this issue [21, 22, 23]. The edge computing platform allows edge nodes to respond to service demands which results in reduced bandwidth consumption and network latency. Figure 1 shows a basic edge computing architecture.

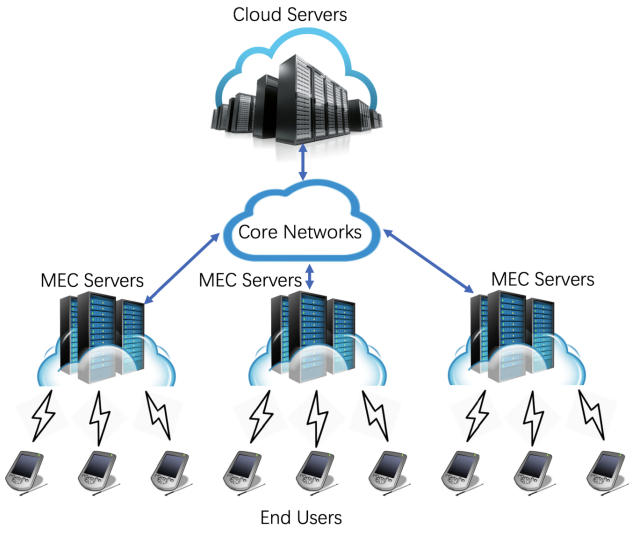

Figure 1: The basic edge computing architecture [1]

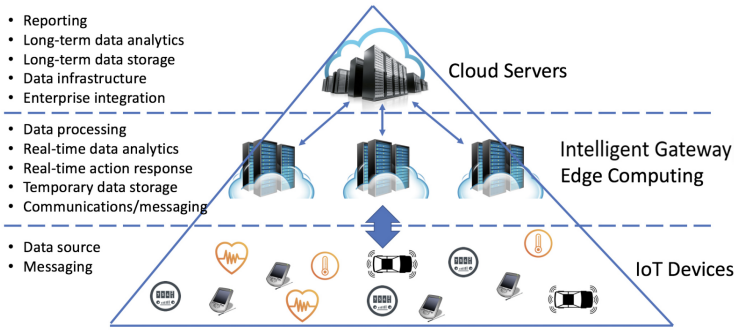

Figure 2: Edge computing based IoT architecture [1]

Edge computing based IoT helps solve some critical issues and improves performance. Not to mention, IoT and edge computing share some characteristics which is further clarified by figure 2.

It further illustrates that IoT devices are end users for edge computing and IoT can benefit from both cloud and edge computing. The latter helps with faster response times and provides a tolerable computational capacity and storage space. Nowadays, the physical end user devices such as smart phones are considered edge devices. They may have an exclusively local component of an application but most

generally, some sort of data is communicated with the cloud server, with or without the help of a gateway device.

One of the restricting factors among IoT edge devices is limited battery life (among limited storage and other things). Even though IoT and mobile devices are enabling a connected future that promises huge amounts of time and money saved with better automation, control in industry and our everyday activities, as well as other benefits such as better health care via remote monitoring and efficient fuel usage in smart and increasingly autonomous vehicles [2], it is a matter of fact that amongst these are portable devices which require a battery to operate. Usually, these devices have small form factors which limit the size of battery that can be used in these devices. There haven't been dramatic improvements in terms of alternative, more reliable battery types, which in turn limits the capabilities of components in such edge devices. Figure 3 shows a breakdown of average energy usage across 1520 users of Samsung Galaxy S3 and S4 mobile devices [2]. In this experiment, users were divided into five groups based on their activity levels. It shows that each group of users has varying needs for which respective hardware component consumes more energy. In the recent years, there has been a strong motivation to minimize energy and power across all of these components so that mobile and IoT devices last longer on a single charge, and to also allow more sophisticated components such as CPU, GPUs and NPUs [2].

This need of minimizing energy consumption of such devices had lead to a plethora of techniques being developed to reduce the overall energy consumption of edge devices. Researchers have used machine learning approaches to categorize applications during execution and choose a suitable pre-existing power plan [24, 25, 26]. However, it was later realized that this does not capture dynamic workload variations [27]. Later on, a new technique was proposed which involved phase level instrumentation to collect workload statistics at runtime. Essentially, the

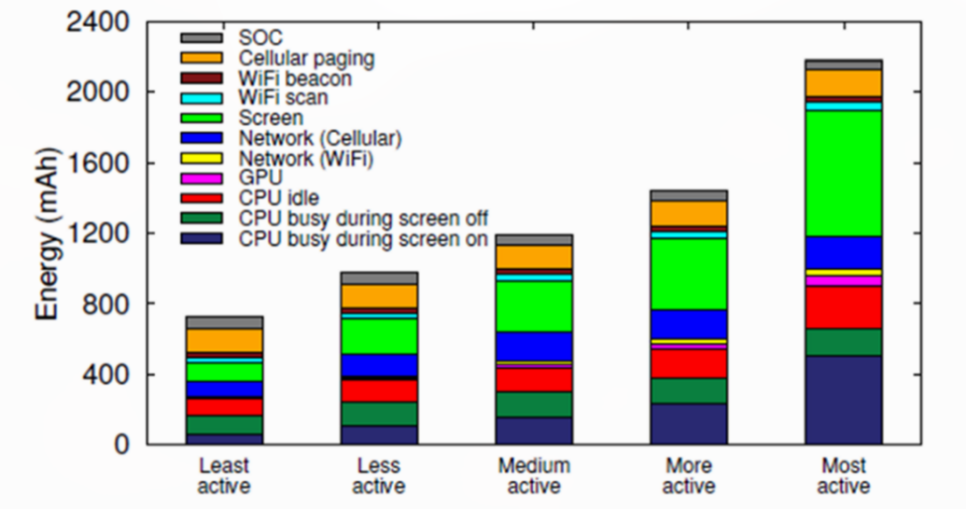

Figure 3: Average daily energy drain breakdown of 5 groups of 1520 apps [2]

workload was divided into snippets and performance application programming interface calls were inserted between each snippet. The data collected from each snippet was then used to control the power states of processing elements [28]. There are other ML based approaches as well where some are better than the other using Reinforcement Learning and Imitation Learning [29, 30, 31, 32, 33, 34, 35]. Other than display optimations [36, 37, 38], wireless radio optimizations [39, 40, 41, 42, 43, 44] and main memory optimizations [45], there have also been software and user-centric optimizations. A variety of OS level energy management techniques have been proposed in literature [46, 47]. Runtime software profiling can identify the most power and energy hungry resources that should be targeted by dynamic management techniques. Powerscope [48] is one such example of a tool. It maps energy consumption of active applications to program structure. It further combines hardware instrumentation to measure current levels with kernel software support to perform statistical sampling of system activity. For mobile devices, the display is the main user interface. User satisfaction is directly impacted by brightness and content on the display. CAPED [3] uses an online learning algorithm

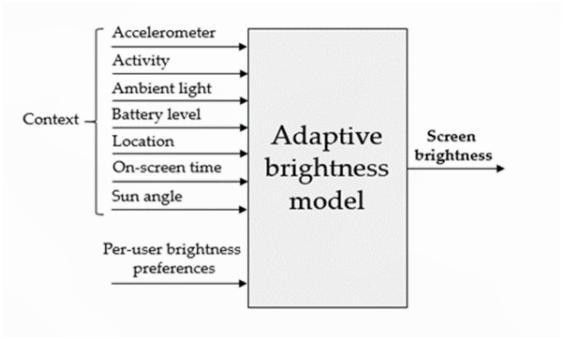

Figure 4: Online learning control of display brightness [3]

that dynamically controls the display brightness to meet individual user's preferences. Figure 4 depicts how CAPED's online learning control works.

Virtualization is commonly defined as the act of creating a virtual version of something, including but not limited to, hardware platforms, storage devices and network resources. It has many advantages including flexibility, capacity, processing power, growth in demand, and energy efficiency [49]. A virtual machine instance represents an entire isolated environment. Multiple isolated environments can run and multiplex the resources of the same host. However, a new model has emerged which has been quite in demand for the past couple of years. *Containers* [7] allow multiple user-space instances and are generally lightweight. They have limited overhead compared to Virtual Machines (VMs) and unlike virtualization, paravirtualization or full virtualization, they do not require an emulation layer to run. Instead they use the host OS [50]. One such notable example of a container framework is Docker [7]. It unarguably started the container movement. Docker is an open source system that provides an automated way to deploy applications inside containers [49].

Docker consists of multiple core components, described quite aptly by [7]:

- **Docker client and daemon:** the client is basically a command-line binary which performs requests for the daemon. The daemon itself can either be run

remotely or on the same host.

- **Docker images:** labeled as the "source code" for the containers, images are the building blocks of Docker.
- **Registries:** used to store the images built. There are two types of registries, public and private
- **Docker container:** applications and services are packaged inside containers. They are released from images and may contain one or more services.

Container performance is an important attribute to evaluate the quality of offered service. It represents system throughput, responsiveness, resource utilization, response time, latency, failure rate and fault tolerance [51]. With containers being very flexible and having the ability to fine tune specific metrics, the important question to ask is whether they can help preserve energy in certain scenarios. This will be further discussed in a later section.

The usage of actors in concurrent programming is a common occurrence. Actors are concurrent objects that communicate by sending each other messages [52]. In the recent years, another framework called Lingua Franca, has been proposed and developed. Lingua Franca or LF, is a polyglot coordination language for concurrent, time sensitive applications which can range anywhere from low level embedded code to distributed cloud and edge applications [53]. An LF program consists of reactors and reactions. Reactions define the functionality of the specified reactors in target languages. Reactors are similar to actors in a sense that they are used to send messages to each other. However, these messages are unique in a sense that they are timestamped and the reactors are deterministic by default. Any non determinism has to be explicitly specified within the LF program. It must be noted that LF is a coordination language and not a programming language, and is only used to write interfaces and composition of reactors [53]. Going into additional details about the

inner workings of LF and its components is out of scope for this study however, the general abstraction of an application's functionality regardless of the target language is something that is similar to the Apache Cordova Host [54]. One question one would like to ask is whether code generated by LF performs better than native code in terms of energy consumption. This too, like Docker, will be discussed in a later section with the help of a case study.

III. RELATED WORK

This section will discuss some of the related work to this thesis. There has been an extensive amount of work on energy optimization - both in and outside of IoT environment. Xiao et al. [8] compared 3G and Wi-Fi technologies in terms of energy consumption. It answered some of the very first basic questions as to what are the factors that influence energy consumption? Therefore, it compared progressive download vs traditional TCP download-and-play, WCDMA vs WLAN 802.11g and phone memory vs external flash drive vs cache. It answered some important questions such as WLAN being more energy efficient than WCDMA, and the energy consumption of youtube video playback being dependent on the amount of video being loaded into cache. However, it failed to adapt to network conditions. Moreover, it was more about the comparison of resources used for video streaming as a whole rather than just the communication aspect of it. Our work on the other hand compares energy consumption under the same communication protocol i.e. TCP/IP for different chunk sizes.

Balasubramanian et al. [9] compared GSM, 3G and Wi-Fi in terms of energy consumption and for each of them, developed a model for energy consumed by network activity. Using this model, they developed TailEnder, which was a protocol to reduce energy consumption for mobile phone apps. The main idea behind it was to schedule tasks that could tolerate a little bit of delay to reduce the overall cumulative energy expended. It also concluded that Wi-Fi was the most efficient of them all for different data transfer sizes as compared to 3G. However, it did not account for the differences between energy usage of different data transfer size within the same paradigm. Our work compares energy consumption of different data transfer sizes among other things.

Gupta et al. [10] conducted a measurement study which provided a detailed

anatomy of power consumption in WiFi phones that can be exploited in designing schemes to prolong the battery charge cycle. It also showed that power saving mode for Wifi together with intelligent scanning techniques can reduce overall energy consumption. The inference drawn from this work was that scanning algorithms have a significant impact on the battery life of WiFi enabled devices [10]. However, this thesis is more geared towards optimizing the energy cost of communication after the connection has been established and data transfer needs to take place.

Sahin et al. [12] argued that even though historically, software developers leave concerns about power consumption to lower-level engineers, it is possible that involving software developers to participate in that process may result in more efficient applications. After detailed evaluation, they concluded that design patterns do impact energy consumption, and that impact within a category is inconsistent. It addressed the lack of tools and techniques to monitor power as an issue. Our work addresses the issue by presenting an energy estimation approach as well as builds upon their argument by helping software developers choose an optimal chunk size for a more energy efficient design process.

In another work, Sahin et al. [13] discuss the impact of code refactoring on energy usage. With the help of an empirical study, they found out that code refactoring can not only impact energy usage but also improve or deteriorate the energy efficiency of the whole program. The motivation of this thesis on the other hand is not really code refactoring but optimizing energy consumption of the WiFi adapter by figuring out a more optimal value for specific variables for certain API calls i.e. data transfer size for socket or HTTP APIs.

Samir et al. [14] worked on energy consumption for different Java Collection Classes. They found out that different classes have different energy blueprints attached to them. In some cases, using an ArrayList is better than a LinkedList while in others, the former would incur a 300% increase energy consumption due to

unoptimized memory accesses and operations. Our work is more about what chunk size is better for which edge device so as to use the least amount of energy. It's more about studying what variable values in a program affect the hardware usage of a physical device in what manner.

The most related work is by Friedman et al. [11] where they studied the effect of throughput on power in a performance study. They compared Bluetooth with WiFi and their usage in smartphones. The study was concluded with the result that power consumption is generally linear with the obtained throughput and overall, Bluetooth uses less energy than WiFi but ultimately its a trade-off between range and efficiency. It also showed that different hardware and different software produce different results for the same experiment, and there is no trend. One of the more interesting conclusions was that there exists an upper bound which is bottlenecked by the receiver (of throughput), when exceeded, would result in more power consumption than not. The additional consumption is explained by the fact that the sender has to retransmit the packets if the packet size is bigger than the receiving window. Our work on the other hand, is different in a sense that we build upon the conclusions of this work and show that even if there is no trend, there is a way to calculate the optimal chunk size (i.e. throughput data size) without knowing the detailed specs of the receiver. We also show that energy usage versus chunk size is also not linear.

Ngu et al. [55] suggested that the use of an IoT middleware framework like Apache Cordova Host [54] to overcome the drawbacks of cloud-based computing frameworks could help improve performance in terms of local processing. Along with the increased portability, the results showed that services deployed using the Cordova Accessor Host were 35% more energy efficient than the native application. Even though our work focuses on reducing energy consumption as well, we focus more on individual variables that affect energy usage rather entire services.

In the next section, we will discuss the methodology employed to estimate energy consumption and determining optimal chunk size in our approach.

IV. METHODOLOGY

As mentioned in the previous sections, the rapid increase in IoT devices as well as its use-cases has significantly impacted and at times more often than not, improved our experiences. However, when these personal, and usually hand-held or wearable devices like smart phones and smart watches, have a plethora of added functionality, it also poses the problem of increased energy consumption. This results on a less uptime for one charge of a battery which corresponds to the device not being utilized for proper amounts of time. This is especially important in the healthcare sector where a device monitoring a patient's vitals should be able to do so for prolonged lengths of time.

In this thesis, our study focused on optimizing energy consumption for communication. Most applications on an edge device usually communicate with a remote server to exchange different data and it varies from application to application. But within these applications, communication is a core aspect which is not being investigated by researchers. The best general practice used when designing these applications is to send data in size multiples of 2, which generally means faster communication without unwanted packet drops but on the other hand, is no where near optimal. This is due to the fact that each edge device may have different hardware and different software versions, and the energy cost for communication for each possible variation of these is different [11]. It has no optimal trend, therefore, the best practice is to use general values of chunk sizes that the protocol can easily handle. Our work focuses on communication cost because most of the times other hardware related factors are not flexible or would otherwise compromise the integrity of the task.

This work is divided into two portions. First, we present a pretty straightforward way to estimate an application's energy consumption over time.

Next, we present an approach to compute optimal chunk sizes for respective edge devices. We evaluate and confirm whether this optimal chunk size approach consumes less energy or not with the help of our energy estimation approach, on two real-world applications.

Estimating Energy Usage

There have been other approaches in the past to estimate energy consumption of applications [56] but we adopt an approach that is similar to energest [57] which is an energy profiling approach for Contiki OS. We combine runtime information from applications with static information from data sheets of specific hardware components to estimate energy consumption. Before getting into the details, we will define some terminologies.

Definition 1 ***Current*** *(I) is defined as the flow of electrical charge (q) over time (t). It is measured in Amperes (A).*

Definition 2 ***Power****(P) is defined as the amount of charge (q) moved through voltage (V) in a time interval (t). It is measured in Watts (W).*

For devices powered by a battery, both current and voltage is an important factor and is heavily dependent on the components used on the device itself. Different electrical components operate on different voltages, thereby, drawing different amounts of current, resulting in different power consumption. However, in order to calculate the energy consumption, the dynamic factor is time. Power used or spent over a specific period of time equals the energy used by the specific device or specific component of device.

Definition 3 ***Energy****(E) is defined as the amount of power (P) expended over a time interval (t). It is measured in Joules (J).*

We use these definitions to be able to express how we estimate the energy consumption of a specific edge IoT device. Almost all hardware components when designed and brought to market have a corresponding data sheet publicly available, with important information such as circuit diagrams, and current and voltage values for different modes (if possible). Current and voltage are dynamic factors however by using information from said data sheets, we can estimate these values by computing the mean average sum of values corresponding to different operating modes. The data sheet for BCM43455, the wifi adapter used in Raspberry Pi 4 Model B [58] contains valuable information regarding different technical details such as frequency, circuit design, power management etc. However, what is interesting to us is current and voltage values. Figure 5 provides a snapshot from one such table. It outlines the different current values corresponding to different voltage values for different modes. In a real world setting, the devices switch between these modes depending on various factors. A reasonable assumption in this case would be to take the mean average sum of values for different modes.

Given this information, we can estimate the power `P` that the wifi adapter would use in this case when it is turned on. Consider an app `A` which finishes executing in about `t` seconds. The total energy consumed (by the wifi adapter) by running `A` would then be calculated by `E = P * t`.

In order to calculate the estimated energy for a specific block of code, such as ones which handle communication, all one needs to do is instrument the app to log execution time for that specific block of code. And then we can calculate the energy usage of that specific portion of program. The main idea behind this approach to estimate energy is that software developers don't usually have the technical equipment required to individually measure current and voltage values of actual devices. Given that setback, estimating energy consumption off of publicly available data sheets and runtime information from the app itself is a pretty reliable way to

Mode	V_{BAT} = 3.6V, V_{DDIO} = 1.8V, T_A25°C	
	$V_{BAT, mA}$	$V_{IO, uA}$[a]
Sleep Modes		
Radio off [b]	0.006	5
Sleep [c]	0.020	200
IEEE Power Save: DTIM = 1, single RX [d]	1.25	200
IEEE Power Save: DTIM = 3, single RX	0.45	200
Active RX Modes		
Continuous RX mode: MCS7, HT20, 1SS [e, f]	55	60
CRS: HT20 [g]	50	60
Mode	V_{BAT} = 3.6V, V_{DDIO} = 1.8V, T_A25°C	
	$V_{BAT, mA}$	$V_{IO, uA}$[a]
Active TX Modes – Internal PA		
Continuous TX mode: 1 Mbps @ 21.5 dBm [h]	400	60
Continuous TX mode: MCS7, HT20, 1SS, 1 TX @ 19 dBm [h]	350	60

a. VIO is specified with all pins idle (not switching) and not driving any loads.
b. WL_REG_ON and BT_REG_ON are both low. All supplies are present.
c. Idle, not associated, or inter-beacon.
d. Beacon Interval = 102.4 ms. Beacon duration = 1 ms @ 1 Mbps. Average current over 3× DTIM intervals.
e. Duty cycle is 100%. Carrier sense (CS) detect/packet receive.
f. Measured using packet engine test mode.
g. Carrier sense (CCA) when no carrier present.
h. Duty cycle is 100%.

Figure 5: Current and Voltage values for BCM43455

gather the required data.

Calculating Optimal Chunk Size

As mentioned before, our study focuses on optimizing communication cost of applications for IoT edge devices. Even though the best practices involved in this case do perform well for average case, our hypothesis is that after a communication pipe is established between the client and the server, having data sent or received in different chunk sizes could have an impact on the energy consumption of the whole process. The motivation behind this is that even though hardware optimizations such as power saving modes improve energy consumption efficiency, there are still software related configurations that can be altered to reduce the overall energy usage. We chose to focus on communication cost because it is an integral part of more or less every app deployed on an edge device where it communicated with the server one way or another to transmit some sort of data.

But why communication cost? Indeed, there are multiple code related factors that can help optimize energy consumption but our work is done under the assumption that these factors, such as data collection, via the sensor, the disk I/O operations etc. differ heavily from use-case to use-case and any alteration in code functionality to them, may affect their usage in real world. For example, a device monitoring a patient's heartbeat continuously is bound to use more energy than, let's say, monitoring every five minutes. However, in these kinds of critical health monitoring applications, such types of optimizations severely undermine the actual functionality of the app. Therefore, our approach addresses a factor that is independent of the sensor usage in such a context and can potentially work on diverse examples.

To better explain how we calculate the optimal chunk size, we will first define energy-per-byte or EPB. It is a metric we use to determine which chunk size consumes the least amount of energy.

Definition 4 ***EPB*** *or Energy-per-Byte is defined as the amount of energy consumed when a single byte of data is sent from one device to another.*

But why is EPB an important factor rather than the overall energy consumed for the corresponding chunk size? We will explain this with the help of an example. Consider an app `C` on the edge device `EdgeD` that needs to send 256 bytes to the server-side app `S`. Assume you calculated the energy to send data at a chunk size of 64 bytes was the lowest at `E1` Joules. However, if you used 128 bytes at a time to send this data, you consume `E2` Joules which is slightly more than `E1`. In this case, we can not say the 64 bytes is the optimal chunk size for `EdgeD` because it would take `E1*4` amounts of energy to send the complete data and `E2*2` on the other hand. For the sake of this example, assume `E1` has the arbitrary value of 2 Joules (actual values are way way less and are usually in microJoules) and `E2` has the arbitrary value of 3 Joules. 64 bytes would take 8 Joules in total to send 256 bytes

of data where as 128 bytes of chunk size would only use 6 Joules. That is why, EPB is an important factor and metric in this case. So what we need to do is divide `E1` by 64 and `E2` by 128 to get the EPB for their respective chunk sizes. Now, the lower energy consumption value of the two would be the optimal one as it would consume the lowest amount of energy among different chunk sizes.

To calculate the optimal chunk size on an edge device, we deploy a Python script which comprises of a list of different chunk sizes of order 2^n where `n` by default can be anything between 1 and 30 (included). The upper limit can be adjusted depending on the use case of the program. It then proceeds to send data in different chunk sizes and then measure the time it took for the communication to take place. This time is then used to calculated the overall energy consumption for sending a specific amount of data. It must be mentioned here that the chunk size is also bounded by the protocol being used and the local disk space of the device. It is provided as a pseudocode in algorithm 1 on the next page. For both evaluation subjects, we use the same pseudocode but had to implement it in Python and Java for both applications respectively.

This is a pretty straightforward but novel approach to aid programmers in developing applications for a target edge device where they can optimize the communication cost in terms of energy. In some use cases, the difference is considerable while in others it may not be that much. In short, it depends on the nature of the app itself. The following section will clarify more as to why it is as we describe our experimental and nature of the applications, and discuss our results.

```
/* N is the upper limit of chunk size as exponent of 2, P is the
   power computed from datasheet                                  */
```
Data : $N \geq 0, P$
```
/* Array of energy consumption for each chunk size                */
```
Result : $energyArray$
$energyArray \leftarrow \phi$;
while $N \neq 0$ **do**
```
    /* Create a byte array of the given size using a helper
       function                                                   */
```
    $array \leftarrow CreateByteArray(2^N)$;
```
    /* Start a timer                                              */
```
    $start \leftarrow GetCurrentTime()$;
```
    /* Send the data to the server                                */
```
    $SendToServer(array)$;
```
    /* Stop the timer and compute the difference                  */
    $end \leftarrow GetCurrentTime()$;
    $executionTime \leftarrow end - start$;
```
    /* Compute the EPB and store it in array   */
```
    $energy \leftarrow P * executionTime$;
    $EPB \leftarrow energy/2^N$;
    $energyArray \leftarrow energyArray \cup EPB$;
    $N \leftarrow N - 1$;
**end**

**Algorithmus 1 :** Pseudocode for determining optimal chunk size
```

V. EVALUATION

In order to evaluate the efficiency of our approach, we evaluated it on two real-world applications. These apps are on completely different hardware with a completely different framework and different use cases. We will present and discuss our results for both of them individually by first explaining the experimental setups involved and then provide reasoning for the results obtained.

Experimental Setup —TempSens

For our first experimental setup, the application we use is called `TempSens`. It is a mock app that was developed from scratch. It continuously monitors and log sensor data from a DHT11 sensor which is used to monitor temperature and humidity levels. Overall, the app is very simple and has a basic functionality: continuously monitor data and communicate it with the server every 60 seconds. The hardware configuration in this setting utilizes a Raspberry Pi 4 Model B running a headless Raspbian OS as an edge device where it functions as a client, and a remote server deployed on a MacBook Pro running Big Sur.

Raspberry Pi 4 Model B houses a System-on-Chip (SoC) which makes it much more affordable and reliable in terms of computing capability. It uses BCM2711 as its SoC type which is one of more energy and cost effective alternatives than its previous counterparts. The `TempSensClient` is written in Python 3 and uses RPi.GPIO to interface with the sensors. Figure 6 shows an extension board that is used to connect sensors to the Raspberry Pi. Figure 7 on the other hand provides the naming methods used for WiringPi, Board and the intrinsic pin name for GPIO extension board. For instance, for GPIO27, the WiringPi naming method is 2, Board naming method is 13 and its intrinsic name is GPIO2 [4]. The overall

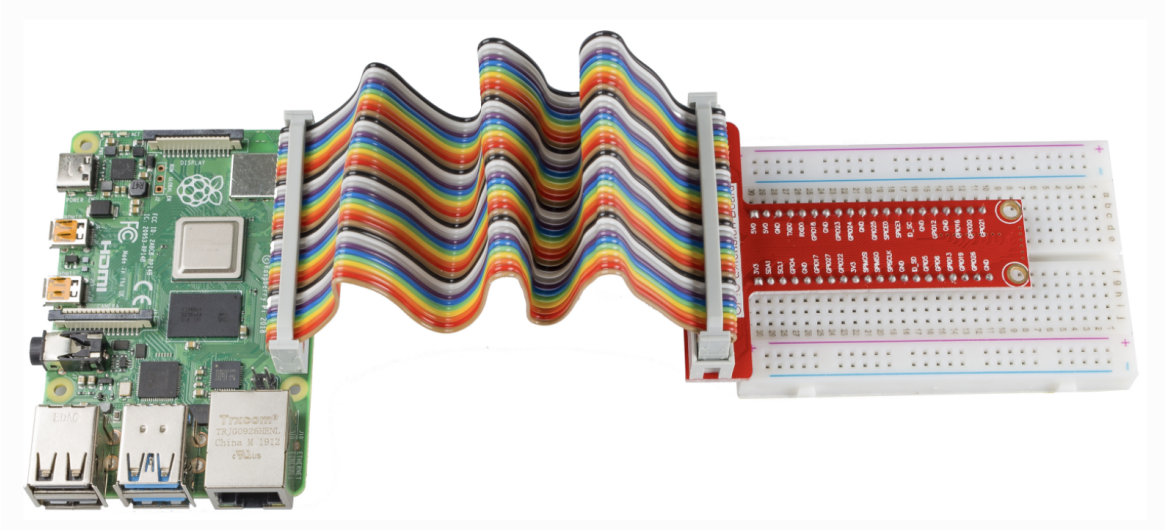

Figure 6: A GPIO extension board [4]

circuitry resembles that in the figure 8. WiringPi [59] is the CPP library for Raspberry Pi to interact with a GPIO extension board and Board [60] is a Python library for the same purpose.

The communication is done over TCP/IP with the help of socket programming. The `TempSensServer` is a bare-bone server which just displays the received information in a proper format. The client collects data for one minute before sending the accumulated data to the server. By default, it uses 512 bytes as chunk size.

Based off Figure 5, the estimated power consumption of the wifi adapter is `1.542` Watts. This is calculated after computing the mean average sum of different power modes for different current values corresponding to the voltage. In order to compute the optimal chunk size, the python script to determine the chunk size was run on the Raspberry Pi 4.

Figure 9 shows that energy consumption does not linearly depend on chunk size. If that were the case, we would have got a straight line. However, what we observe is a global minima and around 2^{14} on the x-axis, we see that's where the

Name	WiringPi	Board	BCM		Board	WiringPi	Name
		GPIO Extension Table					
3.3V	3V3	1	3V3	5.0V	2	5.0V	5V
SDA	8	3	SDA	5.0V	4	5.0V	5V
SCL	9	5	SCL	GND	6	GND	0V
GPIO7	7	7	GPIO4	TXD	8	15	TXD
0V	GND	9	GND	RXD	10	16	RXD
GPIO0	0	11	GPIO17	GPIO18	12	1	GPIO1
GPIO2	2	13	GPIO27	GND	14	GND	0V
GPIO3	3	15	GPIO22	GPIO23	16	4	GPIO4
3.3V	3.3V	17	3.3V	GPIO24	18	5	GPIO5
MOSI	12	19	MOSI	GND	20	GND	0V
MISO	13	21	MISO	GPIO25	22	6	GPIO6
SCLK	14	23	SCLK	CE0	24	10	CEO
0V	GND	25	GND	CE1	26	11	CE1
IN_SDA	30	27	EED	EEC	28	31	ID_SCL
GPIO21	21	29	GPIO5	GND	30	GND	0V
GPIO22	22	31	GPIO6	GPIO12	32	26	GPIO26
GPIO23	23	33	GPIO13	GND	34	GND	0V
GPIO24	24	35	GPIO19	GPIO16	36	27	GPIO27
GPIO25	25	37	GPIO26	GPIO20	38	28	GPIO28
0V	GND	39	GND	GPIO21	40	29	GPIO29

Figure 7: Naming methods for WiringPi and Board [4]

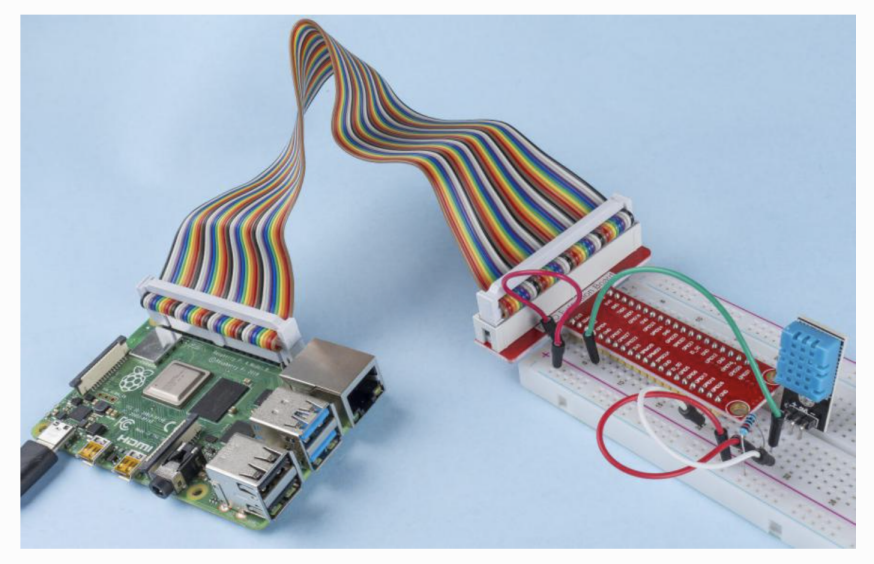

Figure 8: TempSens Hardware Configuration [5]

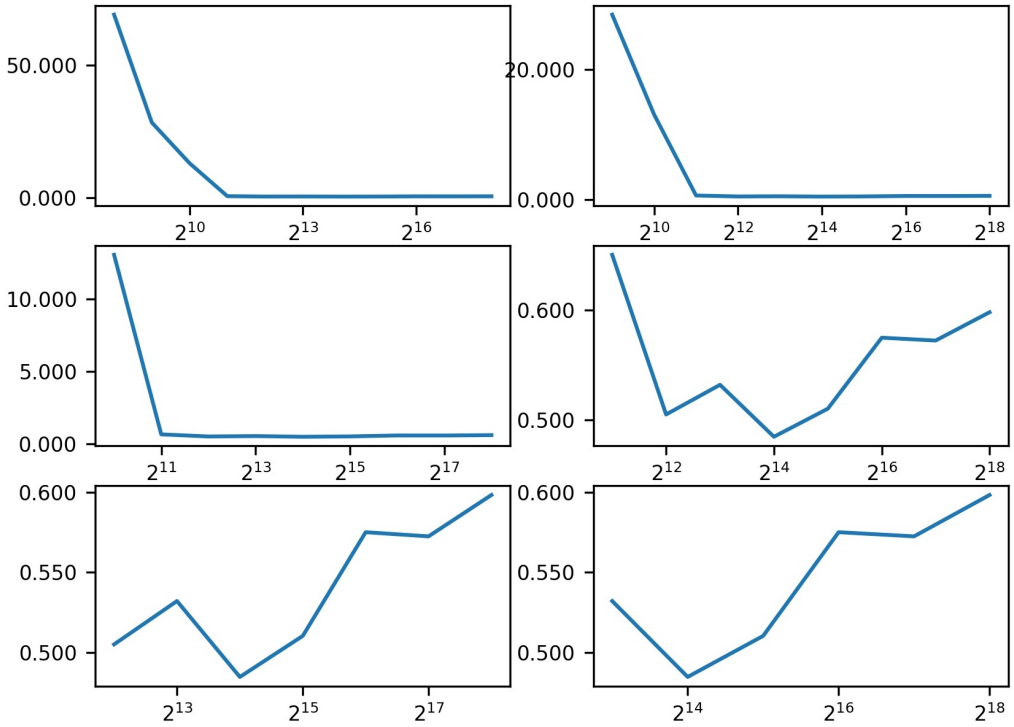

Figure 9: EPB for different chunk sizes (Chunk Size (x-axis) and Energy in microJ (y-axis)) - Raspberry Pi 4

lowest amount of energy is being consumed per byte. This suggests that for the `TempSens` app to have optimal energy consumption for communication, the preferred chunk size should be 2^{14} as it would consume the lowest amount of energy. It can also be seen that EPB at 2^{14} is significantly lower than that at the initial value of 2^9. Figure 9 shows multiple subplots each for a different range of chunk sizes. The idea behind this is to show that even though the energy usage looks pretty much the same when seen in the top-left and top-right plot, but when you actually zoom in and see the bottom-left and bottom-right plots, you realize that the energy usage is not that linear anymore. There's clearly a global minima present and it will be different for different devices due to varying hardware and software configuration but with our work we can definitely find it.

Based on the values obtained from this experiment, and then replace them with the default values in `TempSensClient`, the total amount of time for which battery life was prolonged solely due to saving energy in communication cost was about 3 minutes. However, one other use-case of this approach is to be able to determine the intervals at which one should communicate data, if it allows. In the case of `TempSens`, its quite flexible as the information being communicated is not time sensitive and the use-case is not that critical. This suggests that an optimal interval would be just when the app collects enough amount of data (2^{14}). For this app, it turned out to be 120 seconds which is double the original interval break. Applying this optimization, we were able to prolong the battery time of one charge cycle by 7 minutes. Even though apparently it doesn't sound like much but keep in mind this is just with optimizing communication cost which is usually in terms of microJoules. Comparatively, that is a considerable improvement in the overall scheme of things. On the other hand, it would not have mattered much if the energy usage of the whole device was dependent on a sensor which consumes much more energy as compared to the wifi adapter as we will discuss it in the next experimental setup.

Experimental Setup —Fall Detection App

For this experiment, we test our approach on the Fall Detection App [61]. It is a real-world fall detection app that uses machine learning to predict whether a person wearing a smartwatch has fallen or not. Unlike the app mentioned in [61], the version we test our approach on has majority of the app functionality on the smartwatch itself, whereas the mobile phone is just used to create and load profiles. It uses accelerometer data to predict whether a fall has occurred or not. It continuously logs this information and stores it in a local Couchbase database [62]. The app is written in native Java/Android for WearOS. The app works in a simple manner. First, since the app by default has a generic detection model, the user needs to calibrate it for themselves by wearing the app for around 3 hours. The model is then retrained with personalized accelerometer data for more accurate predictions. The server is maintained on a campus Mercury server [63] which is not publicly accessable to keep confidential data safe. It also houses the training script to retrain the models. The app itself communicates data with this server using HTTP POST. The app uses OkHttp [64] as a communication API between the client and the server.

The smartwatch used is Huawei Watch 2 Sport Model LEO-BX9 running WearOS version 2.14. The partner app to create and load profiles is running on an LG Nexus 5 running Android 8.0. The Huawei Watch 2 Sport uses a Qualcomm chipset housing a WCN3620 wireless connectivity integrated circuit [6].

Figure 10 shows the relevant power consumption information from the WCN3620 datasheet. Just like BCM43455, we can see there are two primary sources of voltage. One of them is V_{xo} or V_{cc} which is the voltage directly supplied by the battery/power source. The other one, $V_{I/O}$ is the voltage that the wifi adapter receives from the CPU indirectly. In order to compute the estimated power, we

Mode	1.8 V I/O	1.8 V XO	VDD_xxx_1P3	VDD_xxx_3P3
WLAN current consumption				
2.4 GHz				
2.4G, 11g, 54 Mbps, 16.8 dBm	0.78 mA	1.15 mA	110.45 mA	174.42 mA
2.4G, 11g, 6 Mbps, 18.8 dBm	0.76 mA	1.15 mA	119.72 mA	203.11 mA
2.4G, 11n, MCS7, 15.2 dBm	0.78 mA	1.15 mA	106.25 mA	149.70 mA
2.4G, 11n, MCS0, 17.8 dBm	0.76 mA	1.15 mA	113.12 mA	180.76 mA
2.4G, 11b, 20.8 dBm	0.68 mA	1.15 mA	111.37 mA	147.34 mA
2.4G, Rx	0.67 mA	1.15 mA	59.06 mA	0 mA

Figure 10: Power Consumption values from WCN3620 Datasheet [6]

calculate the mean average sum of 2.4G, 11g values. We use 11g ones because the protocol standard being used is 802.11g. The other two voltage values are the operating voltages of the circuit and in this context, do not affect the calculations for the estimated power consumption.

To compute the power, we multiply current by voltage for each respective voltage and then take the average. This comes out to be 0.001737 Watts. In order to determine the optimal chunk size, we developed a simple script written in Java and deployed as an android app on the watch to run the experiments and extract a set of values. It uses the same communication API and protocols as the fall detection app. By utilizing the calculated power above, we were able to plot a graph between chunk size and energy consumption based off runtime information of the fall detection app.

Figure 11 shows that the optimal chunk size for the Huawei Watch 2 is 4 Megabytes or 2^{22} bytes. Compare this with the default chunk size being used in the fall detection app of 340 Kilobytes which is about 12 times less than the optimal chunk size. However, this also required altering the communication interval of the app to one hour to have enough data accumulated to be able to send it. This does not interfere with the actual detection functionality of the app. The overall reduction in wifi communication cost by using this chunk size is around 77.97%. In

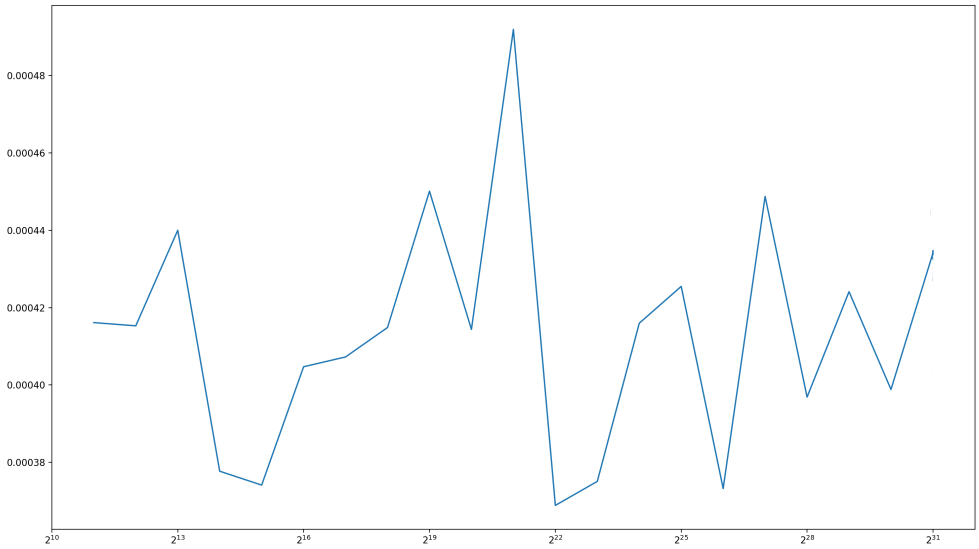

Figure 11: EPB for different chunk sizes (Chunk Size (x-axis) and Energy in milliJ (y-axis)) - Huawei Watch

this context, it is a significant improvement however, in terms of raw battery time saved, it is just around 1.5 minutes. We would also like to mention here that the application failed to upload data to the server when we changed the chunk size to the optimal one. During our initial evaluation while running the Fall Detection application using Android Studio, it was successfully able to write data to the output stream that sends data to the server. Based off the energy estimation readings from that run, the improvement in battery time was around 1.5 minutes. However, when the application was run normally (without Android Studio), it was unable to upload the data to the server even though the data was being sent. There are a number of reasons for which this could be happening. First, the server may have an unintentional bottleneck that we are not aware of. Second, it may be possible that there is not sufficient space on the device to log an hour of data. Lastly, it could be something wrong with the implementation of the application

—particularly how Couchbase is used to store the data in a local database.

Now let's talk about why the improvement is as small as 1.5 minutes. This is due to the fact that Huawei Watch 2 Sport in itself is a highly optimized smartwatch in terms of energy consumption. One of its selling point is its extended battery life due to lower energy consumption. However, our approach is still able to optimize the communication cost albeit only extending the battery life for a few minutes. One other factor that impacts this, is that the core functionality of the app itself is more focused on two things —displaying relevant information on the UI and constantly logging accelerometer data. Moreover, the constant logging is something that can not be altered in this case because the use case of the app is critical in nature. For instance, if we log data in sync with communication i.e. we only read and save sensor data when information needs to be transmitted to the server, it will definitely save more energy but it severely undermines the functionality of the app and ultimately fails to detect some critical fall cases at times when its not logging sensor data. This also shows why our work is more focused on communication cost only as this property is something that will rarely impact the use case of an app.

Moreover, the apps in our study are already somewhat optimized and well-written. Our approach would make even more significant impact if a developer wants to write an app from scratch and then matches the communication interval with the optimal chunk size i.e. when enough data is collected. In the next section, we will discuss a case study on comparing energy consumption of native apps, and apps in a container environment.

VI. CASE STUDY

In this case study, we will discuss how energy consumption is affected if we were to use a containerized environment rather than running an application natively. We will also briefly discuss how Lingua Franca, a polyglot coordination language for distributed programming, may also contribute towards optimizing energy consumption —albeit for only very specific use cases.

Docker

As discussed in Section II, Docker [7] is a container framework which has gained massive popularity and real-world application in the past few years. From applications to distributions, one can run almost anything inside a containerized environment. Figure 12 displays a sample Dockerfile, which is a script to configure the containerized environment. It can be customized as to one's liking and is one of the reasons why Docker has such diverse applications.

For this study, we ran `TempSensClient` as a Docker container on the Raspberry Pi —we will call it `TempSensClientDocker`. The only thing extra that needed to be done was forwarding the port for socket communication when starting up the

```
# syntax=docker/dockerfile:1

FROM python:3.8-slim-buster

WORKDIR /app

COPY helloworld.py .


CMD [ "python3", "helloworld.py" ]
```

Figure 12: A sample Dockerfile

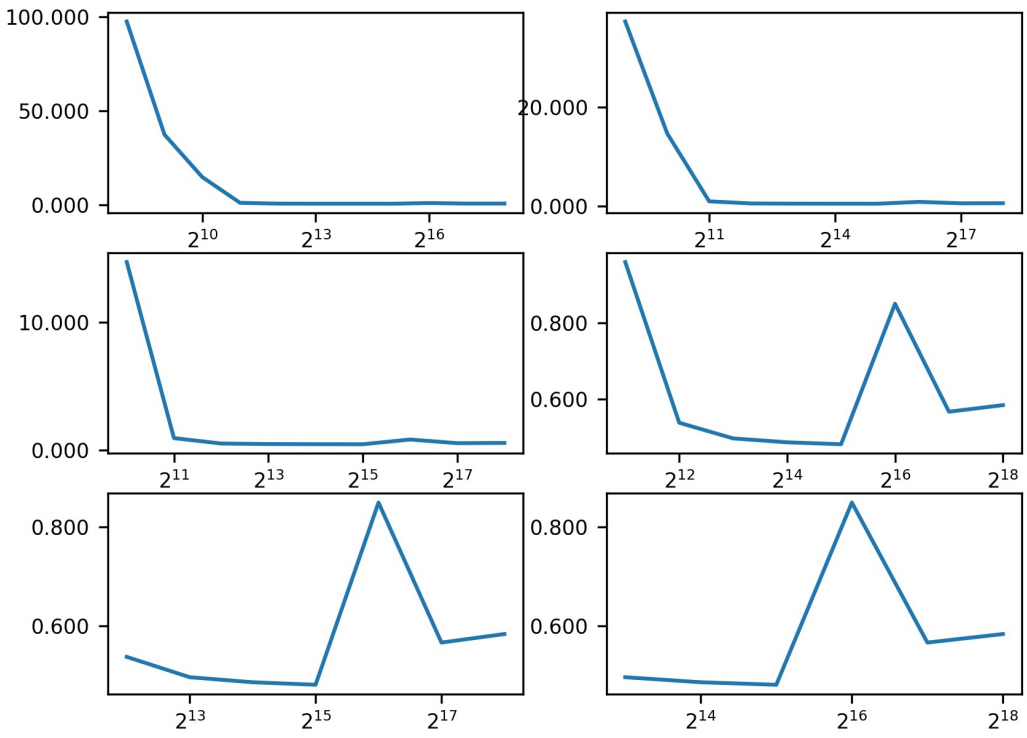

Figure 13: EPB for different chunk sizes (Chunk Size (x-axis) and Energy in microJ (y-axis)) - Raspberry Pi 4 (Docker)

container. This is due to the fact that Docker abstracts the networking layer so the container ports are different from the host ports. The assumption for this study was that `TempSensClientDocker` would consume more energy than `TempSensClient`. However, since we are more interested in communication cost and the energy associated with it, we wanted to compare the communication cost. Figure 13 shows the EPB for `TempSensClientDocker` w.r.t each chunk size.

This experiment shows that the optimal chunk size for `TempSensClientDocker` is 2^{15} unlike `TempSensClient` that had an optimal chunk size of 2^{14}. Even though we did expect a little difference between the energy consumption, what we did not anticipate was getting a completely different optimal chunk size for the same application on the same hardware configuration. Running an application adds an overhead in terms of energy consumption but on the other hand provides more

maintainability and process isolation. However, the different optimal chunk size is not a result of that. As we discussed before, Docker abstracts the networking layer (among other things such as the file system) and in order for `TempSensClientDocker` to be able to communicate with `TempSensServer` over WiFi, we needed to forward the port that the application is supposed to listen at. Recall that in order to compute energy consumption, we need both power and time. The power factor here is static and the same as `TempSensClient` however, the abstraction and port forwarding adds to the time factor and therefore, `TempSensClientDocker` takes a tiny bit longer to communicate as compared to `TempSensClient`, and therefore, has a different optimal chunk size.

One question that we would like to ask is that would it be viable to use Docker to achieve energy efficiency? From our experiment, it is clear that the optimal chunk size is different and the energy consumption corresponding to that is higher than `TempSensClient` as well. We believe that for the sake of running a single application in containerized environment, it is not worth the trade to consume more energy as maintaining one application is a somewhat easier task. However, if multiple applications are running inside the same container, then we could minimize some of the overhead costs and have a trade-off between maintainability and energy consumption depending on the use-case. But, the energy consumption while using Docker would still be greater than running the application on a bare-metal OS. This is due to the fact that containers are based on virtualization and the I/O system calls that interact with the machine as well as abstractions (such as network layer) have a specific overhead. This effect is also discussed by Santos et al. [33] and explains why energy consumption is different in a container environment.

Lingua Franca

Lingua Franca is a framework developed to provide a coordination language for distributed systems. There are a lot of things it is capable of but for the sake of this study, we will focus on features that best relate to our work.

One of the things that was most interesting was how after writing an LF program, the tool produced efficient code for the target programming language. Moreover, it introduces the notion of logical and physical time. Logical time is instantiated the moment the program is executed with the system clock's value, which is just the physical time at that point. However, logical time progresses differently than physical time. Logical time does not advance during the execution of the reaction unlike physical time. Which means that all reactions inside a specific reactor are instantaneous. In other terms, two reactions can run concurrently as long as proper variable value handling is done. Our assumption was that this could mean the code will execute faster (if possible) given the necessary variables had their values instantiated at all possible times, or if the wait time was very minimal. This would further help in reduced energy consumption if the program took lesser than usual execution time (with the one time overhead of writing the LF equivalent and generating the application code). The code below [65] is an example of concurrent reactions being run simultaneously and how the second reaction will print the same result but the process is dependant on first reaction's output. To put it simply, if `out` is set before `out.is_present` is called in the second reaction, it will double the output and print 42. Otherwise, it will just print 42 (from the `else` condition) since there is no logical information available to determine the wait time in this example. This also show-cases how we can improvise and avoid non-determinism. But how much of this actually helps when the application in question is sequential and pretty straightforward in question? How does the efficient

code generation impact energy consumption?

```
reactor Source {
output out;
preamble {=
    import random
=}

reaction(startup) -> out {=
    # Set a seed for random number generation based on the current time.
    self.random.seed()
    # Randomly produce an output or not.
    if self.random.choice([0,1]) == 1:
        out.set(21)
=}
reaction(startup) -> out {=
    if out.is_present:
        out.set(2 * out.value)
    else:
        out.set(42)
=}
}
```

We implemented `TempSens` purely as an LF application with `TempSensClientLingua` and `TempSensServerLingua`. The EPB results were similar to Figure 9 because the Python source code generated by LF from the above versions was about 99% identical to our original version. This is due to the fact that our application was very simple in nature and also had none of the distributed elements that LF could take advantage of. However, we implemented a basic matrix multiplication program in Python utilizing parallelism via multi-threading, and then created an LF version for it (see Appendix A). The LF generated version was 1.742 seconds slower than the original Python program. There are a number of reasons behind this. First, LF has its own overhead because it keeps tracks of logical time among other things. Second, multi-threading is not possible using Python since it is not a thread-safe language. The only type of parallelism that can be obtained is via

multi-processing. And to the best of our knowledge, there is no current LF implementation alternative to pool and collect data from multiple processes.

The motivation behind this study was to determine whether LF generated programs are more energy-efficient or not. From our study, we found out that is not the case with sequential programs with no distributed element to them. The `TempSensClientLingua` took more overall time for communication and hence consumed more energy than `TempSensClient` even though the communication cost was comparable. We believe that since it is designed for distributed applications, it would be able to generate more efficient code. However, we also believe that the communication cost would not be affected by it since it is not a distributed programming construct but since the overall program will be potentially more efficient (due to ease of implementation and understanding through LF) in terms of energy consumption.

We wanted to conduct a study for the Fall Detection application as well but weren't able to do so due to technical limitations. If Fall Detection application were to be implemented as a distributed application, we believe that the source code generated by LF would be more energy-efficient. This stems from the fact that LF programs are highly specific in terms of functionality without the need for any code bloat, and are easier to translate distributed concepts into. This results in less ambiguity and redundancy in the generated code and therefore, with fewer instructions to process, it would consume less energy than its natively written counterpart.

We will discuss the technical limitations in detail in the next section where we will conclude our thesis.

VII. CONCLUSION

In this work, we have proposed a method for estimating energy consumption of edge IoT devices, as well as determining the optimal chunk size for the transmission of data over WiFi that will minimize energy consumption. Utilizing hardware specifications of the component of interest with the help of publicly available data sheets and combining the runtime information of applications, we estimate the total amount of energy consumed for communication. Energy-per-byte or EPB is then used to compute the energy consumption for a range of various chunk sizes. The chunk size corresponding to the lowest energy usage is the optimal chunk size for the transmission of data.

We evaluate our approach on two real-world applications of different natures. Both `TempSens` and the `Fall Detection` application have different use-cases, are deployed on different hardware devices and are developed in different languages. We show that our approach can improve energy utilization of edge IoT devices with regards to communication cost. We discuss the potential reasons why the `Fall Detection` application failed to work and we would like to fix the application issues to make it work in the near future.

We also present a case study where we discuss the possibility of using containers or Lingua Franca, to optimize energy consumption. Even though our evaluation of this was not extensive, we found out that there may be certain situations where the use of containers or Lingua Franca can improve efficiency in other aspects. For example, Docker may shine where maintainability and process isolation is important and the energy overhead it adds helps reduce other workloads such as costs for human-guided maintenance. Similarly, the use of Lingua Franca may help design better and efficient target programs in an easy to use and easy to understand language framework. It may not be the best option for energy optimization but it

does help design distributed programs with fewer chances of concurrency bugs.

However, before we conclude, we would like to discuss some of the technical limitations that we encountered during our study. Firstly, for the Docker case study, we found out that the Docker container is not able to run on an Android device. This is due to the fact the Android kernel does not support Docker as it misses some of the key features, such as namespaces support, of the Linux kernel. Secondly, to the best of our knowledge, there is no container image of Android for Docker that operates on the ARM architecture. For these reasons alone, we were not able to conduct the Docker study for the Fall Detection application. We also wanted to compare communication cost via Bluetooth for `TempSens` but were unable to do due to `TempSensServer` being implemented on a MacOS. The bluetooth bindings for several bluetooth APIs don't properly work since MacOS drivers are proprietary and the best we could do was be able to scan for active devices. Moreover, Docker itself does not abstract and have a virtual bluetooth adapter. It directly uses the host's adapter and the host's bluetooth stack. The hardware device needs to be passed to the container much like the port. However, this means that bluetooth cost would theoretically be the same for the `TempSensClient` and `TempSensClientDocker` versions of the app.

For LF, it currently supports four target languages —TypeScript, C, C++, and Python. Rust support is currently being added and Java support is currently being worked on. Since the current implementation of LF does not support Java, it is not possible to port the Fall Detection application to be implemented in LF. Whether we would get different results or not, the inability and limitation to test this in a much more conclusive manner is a hinderance that we were not able to overcome simply because its not supported yet.

To summarize, this thesis presents an energy estimation and optimal chunk size calculation method for edge IoT devices. We show that this take on energy

optimization has not been proposed before and evaluate it in a real-world setting. Our results show that this approach can improve energy utilization of edge IoT devices in terms of communication cost. In the future, we would like to adapt the optimal chunk size calculation script to automatically compute the estimated power consumption of the wifi adapter given a manufacturer and respective data sheet. For example, for Qualcomm WCN3620 we could parse tables corresponding to "Power Consumption" and for Broadcom BCM43455, we could parse tables corresponding to "WLAN Current Consumption". This is due to the fact that manufacturers follow their own standards for providing data sheets, and consequently, all Broadcom data sheets will provide the current information under "WLAN Current Consumption", and same goes for other manufacturers. This way we have an automated way of estimating the power consumption of an edge device's hardware as well as calculating the optimal chunk size in a single script. Other than that, we would also like to calculate the optimal chunk size for Bluetooth communication and analyze the results.

APPENDIX SECTION

APPENDIX A

```
# Matrix Multiplication -- LF implementation of Python program
target Python;
preamble {=
    import time
    import threading

    def mult(X, Y):
        result = [[0]*100]
        for z in range(len(Y[0])):
            for k in range(len(Y)):
                result[0][z] += X[0] * Y[k][z]
=}
reactor Source{
    output matrices;
    reaction(startup) -> matrices{=
        m = 100
        X = [[1]*m]*m
        Y = [[1]*m]*m
        matrices.set((X, Y))
    =}
}
reactor Mult{
    input m;
    state threads({=list()=})
    reaction(m){=
        X, Y = m.value[0], m.value[1]
        start = time.perf_counter()
        for i in range(len(X[0])):
            mult(X[i], Y)
        end = time.perf_counter()

        print(f"Time: {round(end - start, 5)} seconds")

        start = time.perf_counter()
        for i in range(len(X[0])):
```

```
            x = threading.Thread(target = mult, args=(X[i], Y))
            self.threads.append(x)
            x.start()
        end = time.perf_counter()

        print(f"Time: {round(end - start, 5)} seconds")
    =}
}
main reactor LinguaMatMult{
    src = new Source()
    multi = new Mult()
    src.matrices -> multi.m

}
```

REFERENCES

[1] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, "A survey on the edge computing for the internet of things," *IEEE Access*, vol. 6, pp. 6900–6919, 2018.

[2] S. Pasricha, R. Ayoub, M. Kishinevsky, S. K. Mandal, and U. Y. Ogras, "A survey on energy management for mobile and iot devices," *IEEE Design Test*, vol. 37, no. 5, pp. 7–24, 2020.

[3] M. Schuchhardt, S. Jha, R. Ayoub, M. Kishinevsky, and G. Memik, "Caped: Context-aware personalized display brightness for mobile devices," in *2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pp. 1–10, 2014.

[4] "Gpio extension board." `https://docs.sunfounder.com/projects/davinci-kit/en/latest/gpio_extension_board.html`.

[5] "Gpio dht circuit." `https://docs.sunfounder.com/projects/davinci-kit/en/latest/2.2.3_dht-11.html`.

[6] "Wcn3620 ic." `https://developer.qualcomm.com/qfile/29369/lm80-p0436-33_b_wcn3620_wireless_connectivity_ic_device_spec.pdf`.

[7] J. Turnbull, *The Docker Book*. s.n., 2014.

[8] Y. Xiao, R. S. Kalyanaraman, and A. Yla-Jaaski, "Energy consumption of mobile youtube: Quantitative measurement and analysis," in *2008 The Second International Conference on Next Generation Mobile Applications, Services, and Technologies*, pp. 61–69, 2008.

[9] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani, "Energy consumption in mobile phones: a measurement study and implications for network applications.," pp. 280–293, 2009.

[10] A. Gupta and P. Mohapatra, "Energy consumption and conservation in wifi based phones: A measurement-based study," in *2007 4th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks*, pp. 122–131, 2007.

[11] R. Friedman, A. Kogan, and Y. Krivolapov, "On power and throughput tradeoffs of wifi and bluetooth in smartphones," *IEEE Transactions on Mobile Computing*, vol. 12, no. 7, pp. 1363–1376, 2013.

[12] C. Sahin, F. Cayci, I. L. M. GutiÃ©rrez, J. Clause, F. Kiamilev, L. Pollock, and K. Winbladh, "Initial explorations on design pattern energy usage," in *2012 First International Workshop on Green and Sustainable Software (GREENS)*, pp. 55–61, 2012.

[13] C. Sahin, L. Pollock, and J. Clause, "How do code refactorings affect energy usage?," 2014.

[14] S. Hasan, Z. King, M. Hafiz, M. Sayagh, B. Adams, and A. Hindle, "Energy profiles of java collections classes," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pp. 225–236, 2016.

[15] D. Linthicum, "Responsive data architecture for the internet of things," *Computer*, vol. 49, no. 10, pp. 72–75, 2016.

[16] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, and W. Zhao, "A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1125–1142, 2017.

[17] J. A. Stankovic, "Research directions for the internet of things," *IEEE Internet of Things Journal*, vol. 1, no. 1, pp. 3–9, 2014.

[18] J. Wu and W. Zhao, "Design and realization of winternet: From net of things to internet of things," *ACM Trans. Cyber-Phys. Syst.*, vol. 1, Nov. 2016.

[19] S. Vashi, J. Ram, J. Modi, S. Verma, and C. Prakash, "Internet of things (iot): A vision, architectural elements, and security issues," in *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, pp. 492–496, 2017.

[20] "Interconnected devices via statisa." `https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide/`.

[21] W. Yu, H. Xu, H. Zhang, D. Griffith, and N. Golmie, "Ultra-dense networks: Survey of state of the art and future directions," in *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, pp. 1–10, 2016.

[22] M. Agiwal, A. Roy, and N. Saxena, "Next generation 5g wireless networks: A comprehensive survey," *IEEE Communications Surveys Tutorials*, vol. 18, no. 3, pp. 1617–1655, 2016.

[23] P. Demestichas, A. Georgakopoulos, D. Karvounas, K. Tsagkaris, V. Stavroulaki, J. Lu, C. Xiong, and J. Yao, "5g on the horizon: Key challenges for the radio-access network," *IEEE Vehicular Technology Magazine*, vol. 8, no. 3, pp. 47–53, 2013.

[24] A. Aalsaud, R. Shafik, A. Rafiev, F. Xia, S. Yang, and A. Yakovlev, "Power–aware performance adaptation of concurrent applications in heterogeneous many-core systems:," pp. 368–373, 08 2016.

[25] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack amp; cap: Adaptive dvfs and thread packing under power caps," in *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 175–185, 2011.

[26] G. Dhiman and T. S. Rosing, "System-level power management using online learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 28, no. 5, pp. 676–689, 2009.

[27] P. Bogdan and R. Marculescu, "Workload characterization and its impact on multicore platform design," in *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 231–240, 2010.

[28] U. Gupta, C. Patil, G. Bhat, P. Mishra, and U. Ogras, "Dypo: Dynamic pareto-optimal configuration selection for heterogeneous mpsocs," *ACM Transactions on Embedded Computing Systems*, vol. 16, pp. 1–20, 09 2017.

[29] Z. Chen and D. Marculescu, "Distributed reinforcement learning for power limited many-core system performance optimization," in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 1521–1526, 2015.

[30] F. M. M. u. Islam and M. Lin, "Hybrid dvfs scheduling for real-time systems based on reinforcement learning," *IEEE Systems Journal*, vol. 11, no. 2, pp. 931–940, 2017.

[31] Q. Zhang, M. Lin, L. T. Yang, Z. Chen, S. U. Khan, and P. Li, "A double deep q-learning model for energy-efficient edge scheduling," *IEEE Transactions on Services Computing*, vol. 12, no. 5, pp. 739–749, 2019.

[32] U. Gupta, S. K. Mandal, M. Mao, C. Chakrabarti, and U. Y. Ogras, "A deep q-learning approach for dynamic management of heterogeneous processors," *IEEE Computer Architecture Letters*, vol. 18, no. 1, pp. 14–17, 2019.

[33] S. Ross, G. J. Gordon, and J. A. Bagnell, "No-regret reductions for imitation learning and structured prediction," *CoRR*, vol. abs/1011.0686, 2010.

[34] R. G. Kim, W. Choi, Z. Chen, J. R. Doppa, P. P. Pande, D. Marculescu, and R. Marculescu, "Imitation learning for dynamic vfi control in large-scale manycore systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 9, pp. 2458–2471, 2017.

[35] S. K. Mandal, G. Bhat, C. A. Patil, J. R. Doppa, P. P. Pande, and U. Y. Ogras, "Dynamic resource management of heterogeneous mobile platforms via imitation learning," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 12, pp. 2842–2854, 2019.

[36] X. Chen, J. Mao, J. Gao, K. W. Nixon, and Y. Chen, "Morph: Mobile oled-friendly recording and playback system for low power video streaming," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2016.

[37] M. Dong, Y.-S. K. Choi, and L. Zhong, "Power-saving color transformation of mobile graphical user interfaces on oled-based displays," in *ISLPED*, 2009.

[38] X. Chen, K. W. Nixon, H. Zhou, Y. Liu, and Y. Chen, "Fingershadow: An OLED power optimization based on smartphone touch interactions," in *6th Workshop on Power-Aware Computing and Systems (HotPower 14)*, (Broomfield, CO), USENIX Association, Oct. 2014.

[39] I. Constandache, S. Gaonkar, M. Sayler, R. Choudhury, and L. Cox, "Enloc: Energy-efficient localization for mobile phones," pp. 2716 – 2720, 05 2009.

[40] K. Lin, A. Kansal, D. Lymberopoulos, and F. Zhao, "Energy-accuracy trade-off for continuous mobile device location," pp. 285–298, 01 2010.

[41] R. Krashinsky and H. Balakrishnan, "Minimizing energy for wireless web access with bounded slowdown," *Wireless Networks*, vol. 11, 09 2002.

[42] M.-R. Ra, J. Paek, A. Sharma, R. Govindan, M. Krieger, and M. Neely, "Energy-delay tradeoffs in smartphone applications," pp. 255–270, 06 2010.

[43] B. Kellogg, V. Talla, S. Gollakota, and J. R. Smith, "Passive wi-fi: Bringing low power to wi-fi transmissions," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, (Santa Clara, CA), pp. 151–164, USENIX Association, Mar. 2016.

[44] F. Lu, G. M. Voelker, and A. C. Snoeren, "Slomo: Downclocking wifi communication," in *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, (Lombard, IL), pp. 255–258, USENIX Association, Apr. 2013.

[45] C. Chou, P. Nair, and M. K. Qureshi, "Reducing refresh power in mobile devices with morphable ecc," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 355–366, 2015.

[46] N. Vallina-Rodriguez and J. Crowcroft, "Erdos: achieving energy savings in mobile os," 01 2011.

[47] A. Javed, M. Shahid, M. Sharif, and Y. Mussarat, "Energy consumption in mobile phones," *International Journal of Computer Network and Information Security*, vol. 9, pp. 18–28, 12 2017.

[48] J. Flinn and M. Satyanarayanan, "Powerscope: a tool for profiling the energy usage of mobileapplications," pp. 2–10, 03 1999.

[49] N. G. Bachiega, P. S. L. Souza, S. M. Bruschi, and S. d. R. S. de Souza, "Container-based performance evaluation: A survey and challenges," in *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 398–403, 2018.

[50] R. Dua, A. R. Raja, and D. Kakadia, "Virtualization vs containerization to support paas," in *2014 IEEE International Conference on Cloud Engineering*, pp. 610–614, 2014.

[51] S. Olabiyisi, E. Omidiora, F. Uzoka, and B. Akinnuwesi, "A survey of performance evaluation models for distributed software system architecture," 10 2010.

[52] M. Lohstroh, C. Menard, S. Bateni, and E. Lee, "Toward a lingua franca for deterministic concurrent systems," *ACM Transactions on Embedded Computing Systems*, vol. 20, pp. 1–27, 05 2021.

[53] Lf-Lang, "lingua-franca wiki." `https://github.com/lf-lang/lingua-franca/wiki/Overview`.

[54] "Cordovahost." `https://wiki.eecs.berkeley.edu/accessors/Main/CordovaHost`.

[55] A. H. Ngu, J. Eyitayo, G. Yang, C. Campbell, Q. Z. Sheng, and J. Ni, "An iot edge computing framework using cordova accessor host," *IEEE Internet of Things Journal*, pp. 1–1, 2021.

[56] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 92–101, 2013.

[57] A. Sabovic, C. Delgado, J. Bauwens, E. De Poorter, and J. Famaey, "Accurate online energy consumption estimation of iot devices using energest," pp. 363–373, 11 2019.

[58] "Bcm43455 data sheet." `https://www.cypress.com/file/298786/download`.

[59] "Reference: Wiring pi." `http://wiringpi.com/reference/`, May 2013.

[60] "Rpi.gpio." `https://pypi.org/project/RPi.GPIO/`.

[61] T. Mauldin, M. Canby, V. Metsis, A. Ngu, and C. Rivera, "Smartfall: A smartwatch-based fall detection system using deep learning," *Sensors*, vol. 18, p. 3363, 10 2018.

[62] "Couchbase." `https://www.couchbase.com/about`.

[63] "Mercury txst server." `https://mercury.cs.txstate.edu`.

[64] "Okhttp." `https://github.com/square/okhttp`.

[65] "Lf example." https://github.com/lf-lang/lingua-franca/wiki/Writing-Reactors-in-Python.