6-27-2018

Energy Measurement and Profiling of Internet of Things Devices

Immanuel Amirtharaj
Santa Clara University

Santa Clara University

Department of Computer Engineering

Date: June 27, 2018

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Immanuel Amirtharaj

ENTITLED

Energy Measurement and Profiling of Internet of Things Devices

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

B. Dezfouli

Thesis Advisor
Dr. Behnam Dezfouli

[illegible]

Thesis Reader
Dr. Silvia Figueira

[illegible]

Chairman of Department
Dr. Nam Ling

Energy Measurement and Profiling of Internet of Things Devices

By

Immanuel Amirtharaj

Dissertation

Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Engineering
in the School of Engineering at
Santa Clara University, 2018

Santa Clara, California

Acknowledgments

I would like to thank my advisor Dr. Behnam Dezfouli for all of his help and support over the past year. Your patience, knowledge, and guidance has really helped me through this whole year. Although the work I have done at SIOTLAB has been challenging, it also was the most rewarding. I am thankful that this experience has equipped me with both the skill set and confidence to be successful in the future.

I would like to thank all of the researchers I got to know over the past year. In particular, I would like to thank Chelsey Li and Tai Groot for being great research partners. Collaborating with you was an unforgettable experience and I am glad we were able to accomplish a lot over the past year. I would also like to thank Jay Sheth and Puneet Kumar for being great labmates. Thank you for keeping me company in the lab and for providing me with valuable advice on research and industry. I would also like to thank the different groups who used EMPIOT for their energy measurement experiments. Thank you for notifying me of any bugs and for providing valuable feedback for potential new features.

I would also like to thank my mother Priscilla, and my father Abraham, for all of their encouragement and guidance. Thank you for giving the freedom and support to pursue my interests. Everything I have been able to achieve is because of you.

Finally, I would like to thank my friends, fellow students, and professors for being flexible and supportive with me due to my demanding schedule.

Energy Measurement and Profiling of Internet of Things Devices

Immanuel Amirtharaj

Department of Computer Engineering
Santa Clara University
Santa Clara, California
2018

ABSTRACT

As technological improvements in hardware and software have grown in leaps and bounds, the presence of IoT devices has been increasing at a fast rate. Profiling and minimizing energy consumption on these devices remains to be an an essential step towards employing them in various application domains. Due to the large size and high cost of commercial energy measurement platforms, the research community has proposed alternative solutions that aim to be simple, accurate, and user friendly. However, these solutions are either costly, have a limited measurement range, or low accuracy. In addition, minimizing energy consumption in IoT devices is paramount to their wide deployment in various IoT scenarios. Energy saving methods such as duty-cycling aim to address this constraint by limiting the amount of time the device is powered on. This process needs to be optimized, as devices are now able to perform complex, but energy intensive tasks due to advancements in hardware.

The contributions of this paper are two-fold. First we develop an energy measurement platform for IoT devices. This platform should be accurate, low-cost, easy to build, and configurable in order to scale to the high volume and varying requirements for IoT devices. The second contribution is improving the energy consumption on a Linux-based IoT device in a duty-cycled scenario. It is important to profile and optimize boot up time and shutdown time, and improve the way user applications are executed.

EMPIOT is an accurate, low-cost, easy to build, and flexible power measurement platform. We present the hardware and software components that comprise EMPIOT and then study the effect of various design parameters on accuracy. In particular, we analyze the effect of driver, bus speed, input voltage, and buffering mechanisms on sampling rate, measurement accuracy, and processing demand. In addition to this, we also propose a novel calibration technique and report the calibration parameters under different settings. In order to demonstrate EMPIOT's scalability, we evaluate its performance against a ground truth on five different devices. Our results show that for very low-power devices that utilize 802.15.4

wireless standard, measurement error is less than 4%. In addition, we obtain less than 3% error for 802.11-based devices that generate short and high power spikes.

The second contribution is the optimization the energy consumption of IoT devices in a duty cycled scenario by reducing boot up duration, shutdown duration, and user application duration. To this end, we study and improve the amount of time a Linux-based IoT device is powered on to accomplish its tasks. We analyze the processes of system boot up and shutdown on two platforms, the Raspberry Pi 3 and Raspberry Pi Zero Wireless, and enhance duty-cycling performance by identifying and disabling time consuming or unnecessary units initialized in the userspace. We also study whether SD card speed and SD card capacity utilization affect boot up duration and energy consumption. In addition, we propose Pallex, a novel parallel execution framework built on top of the `systemd init` system to run a user application concurrently with userspace initialization. We validate the performance impact of Pallex when applied to various IoT application scenarios: (i) capturing an image, (ii) capturing and encrypting an image, (iii) capturing and classifying an image using the the k-nearest neighbor algorithm, and (iv) capturing images and sending them to a cloud server. Our results show that system lifetime is increased by 18.3%, 16.8%, 13.9% and 30.2%, for these application scenarios, respectively.

Contents

List of Figures

List of Tables

List of Abbreviations

E_{btl}	Energy of the bootloader phase
E_{knl}	Energy of the kernel phase
E_{sdn}	Energy of the shutdown phase
E_{usi}	Energy of the userspace initialization phase
P_{btl}	Bootloader phase
P_{knl}	Kernel phase
P_{usi}	Userspace initialization phase
T_{btl}	Duration of the bootloader phase
T_{knl}	Duration of the kernel phase
T_{sdn}	Duration of the shutdown phase
T_{usi}	Duration of the userspace initialization phase
ADC	Analog Digital Converter
DMM	Digital Multimeter
EMPIOT	Energy Measurement Platform for Internet of Things
EU	Essential Units
GPIO	General Purpose Input/Output
GPU	Graphics Processing Unit
IoT	Internet of Things
NRS	Networking Related Services
OS	Operating System
PCT	Programmable Calibration Tool
PGA	Power Gain Amplifier

RPi	Raspberry Pi 3 or Raspberry Pi Zero w/ Wireless
RPi3	Raspberry Pi 3
RPiZW	Raspberry Pi Zero w/ Wireless
RSL	Raspbian Stretch Lite
SDC	Secure Digital Card
SIOTLAB	Santa Clara University Internet of Things Laboratory
SoC	System on a Chip
UART	Universal Asynchronous Receiver Transmitter

Chapter 1
Introduction

The importance of low-cost and accurate energy measurement of Internet of Things (IoT) devices can be justified on two fronts. First, the percentage of energy consumed by connected devices is increasing due to the significant growth in the number of IoT devices. Gartner predicts that by 2020, the number of IoT devices will surpass 20 billion [1]. Most of these devices are being deployed in a variety of application domains such as remote areas where there is no stable source of electricity. As a result, they usually rely on some sort of finite energy source such as a battery or energy harvesting solution such as solar power. Therefore, extensive measurement, profiling, analyzing, and improvement of energy consumption is required in order to satisfy QoS requirements. Although there are existing solutions that are currently being used for energy measurement, they are not optimal for the unique requirements necessary for IoT devices. For example, while analytic and simulation based energy estimation tools are simple and easy to use, they fail to take into consideration the complexities of hardware and newer wireless technologies. Although commercial energy measurement platforms provide high accuracy, they are costly, expensive, and bulky; characteristics that do not make them suitable candidates for measuring hundreds of devices at a time.

In addition to energy measurement, the actual conservation of energy in IoT devices remains relevant as ever. Technological advancements, such as increases in processing power and memory capacity have also enabled the next generation

IoT devices for new, process-intensive applications. Such applications benefit from edge and fog computing to improve responsiveness and minimize the overhead of data exchange with cloud platforms [2]. For example, image processing through artificial intelligence on IoT devices presents several benefits including: faster decision-making and lower reaction time to environmental changes, less wireless interference with nearby devices, and improved security due to eliminating the need to upload raw images to a cloud service [3]. According to ABI Research [4], 90% of the data generated by edge devices is being processed locally. Local storage and processing can be used to satisfy the stringent latency requirements of mission-critical applications [5], reduces network utilization, reduces the processing overhead of resource-constraint devices, enhances security, and enables the system to continue its operation even in the presence of intermittent network connectivity [6]. For IoT devices that are connected to a power grid, it is important to reduce their energy footprint given the increasing cost of energy and the efforts towards reducing carbon emissions [7, 8]. To this end, various solutions have been proposed, such as energy-efficient hardware design [9, 10, 11], low-overhead operating systems (OS) [12, 13, 14, 15], and low-power networking stacks [16, 5, 17, 18].

1.1 Low-cost and Accurate Energy Measurement

While there are many different solutions used to measure power consumption for IoT devices they all have their own shortcomings. For example, analytic and simulation-based energy estimation tools are simple and easy to measure power consumption. This is the main reason why most research contributions on low power wireless IoT systems utilize them. Most of these tools function by multiplying the time spent in each state, e.g. sleep, processing, and transmission by the power consumed in that state [19, 20, 21, 22, 23]. Unfortunately this approach is inaccurate as it does not take into consideration the complexities involved in identifying power consumption in each state. First, energy consumption of all operational modes may not be available for a given system-on-chip (SoC). For example, the datasheet of one of these devices may include only the average current consumption for nominal transmission power values. Additionally, the energy consumption of the processor depends on its utilization level, frequency scaling, and I/O operations [24]. Secondly, IoT boards usually include a SoC and several peripheral components which include but are not limited to analog digital converters, sensors, and memory. Therefore, even if energy characteristics of each individual component of the SoC are known, it is still challenging to estimate total energy consumption. Finally, various aspects of software such as code structure, algorithms, and data structures affect the energy consumption of the IoT system. For example, when cache memory is present on a device, an array contiguously allocated in memory has a positive impact on cache performance, which in turn improves energy consumption. Finally, simulations make it difficult to simulate and evaluate the properties of real-world environments, like interference, which make it difficult to study their effects on energy consumption [25, 26, 27]. Despite these shortcomings, most of the research contributions on low-powered wireless IoT systems rely on simulation due to simplicity.

Another well known solution for power profiling is the use of commercial energy measurement platforms. They provide a high degree of accuracy which makes them a popular choice for serious power measurement. Unfortunately, these platforms are either costly, bulky, or difficult to integrate with IoT devices. For example, the Keysight 14465A [28] costs more than $1300 with 2MB storage and a maximum sampling rate of 50Ksps. Similarly, the Keithly 7510 [29] costs more than \$3500 with 2MB storage and sampling rate up to 1Msps. In addition to their size and cost, these devices are not a viable solution towards monitoring energy consumption of a large number of IoT devices in a testbed. Furthermore, their programmability is limited and inflexible which makes it difficult to test certain stages of an operation on a device. These shortcomings also apply to the Monsoon power meter, an \$800 platform widely used in academia. Due to these reasons, using commercial energy measurement platforms are not ideal for the measurement of power consumption in IoT devices.

In response to these shortcomings the research community has also proposed a few cost-effective and accurate power measurement platforms. Based on their main shortcoming, we categorize these platforms as the following: (i) complex (ii) limited supported range (iii) low accuracy. Solutions with complex circuitry make the device costly and difficult to build. Energy measurement platforms with limited supported range are not ideal for measuring energy of IoT devices, especially newer ones that use wireless technologies such as 802.11 which result in current spikes as high as 700mA. Finally, if the accuracy of an energy measurement platform is low, it cannot be used for effective study, development, and debugging of IoT devices. In addition to these shortcomings, most of these platforms ignore the effect of voltage variation on energy measurement [30, 31, 21, 22]. Furthermore, in terms of accuracy analysis, evaluations are very limited and mostly include one IoT device type [32, 33, 31, 21]. We further elaborate on each of these issues in Section 2.1.

In Chapter 3 we propose EMPIOT (Energy Measurement Platform Internet of Things), an accurate, low-cost, easy to build, and flexible power measurement platform designed to solve the problems highlighted earlier. It has two major components - a custom shield built with the INA219 [34] energy monitoring chip, and a base board which mounts and communicates with it. The shield is responsible for performing both current and voltage measurement and costs less than \$5. Depending on the configuration used, INA219 can measure currents as high as 3.2A and voltages up to 32V. EMPIOT's software is a multithreaded C++ program which reads data from the shield and saves it to file. In addition, it supports a set of features that make it easy to control and integrate with existing testbeds. While EMPIOT is simple in terms of hardware, we have extensively experimented and analyzed various design parameters on performance. We profile and evaluate two I2C drivers - the BCM2835 [35] and the Linux driver [36] and show that by using the BCM driver we can attain higher speed lower energy consumption, and predictable timing. In addition, we also evaluate the effect of input voltage on sampling rate and measurement accuracy. Our results show that reducing the shield's operational voltage results in an increase in conversion time which leads to a decrease in sampling rate. Through experimenting with a battery, external power source, and a Raspberry Pi we confirm that EMPIOT can run on variable power sources without affecting accuracy which always falls between 100μA and 4mV. We also compare sampling rate and energy efficiency of the base board using two different data structures for data collection - a single circular buffer and two linear buffers. We observe that using the circular buffer results in a faster sampling rate while using two buffers results in lower energy consumption. Finally, we study the effect of batched and continuous file writes on energy consumption of the base board. We study the effect the driver had as well. By using the BCM driver and batched write operations we are able to considerably reduce energy consumption.

In addition to high performing software for collecting samples, we also need to calibrate the platform to improve accuracy. In order to calibrate the platform, we designed a *programmable calibration tool* (PCT), which allows us to generate currents and voltages in a wide range and precisely control the duration of each change. Using PCT, we calibrate EMPIOT for currents up to 800mA. We study the accuracy of EMPIOT by using five different IoT devices, and four types of loads. Our results confirm that measurement error is less than 4% for very low-power devices that use the 802.15.4 wireless standard and generate peak currents up to 30mA. In addition, the error is less than 3% for 802.11 devices whose current consumption surpasses 10mA. We also show that neglecting voltage variations in the energy measurement process may result in up to 0.5% increase in measurement error, especially for battery-powered 802.11-based IoT devices.

While we never intended for EMPIOT to be a complex power measurement tool, we study and assert that it or even the original INA219 breakout board can be used for accurate power measurement of a wide range of IoT devices. In addition, the total cost of one EMPIOT unit is $35 which makes it an economic and scalable choice to measure power consumption for a wide range of IoT devices.

In addition to being accurate and low cost, EMPIOT offers a high degree of flexibility, supporting a variety of features and integrating easily with different testbeds and IoT devices. EMPIOT been also validated through its use by various groups in the Santa Clara University Internet of Things Laboratory (SIOTLAB) for various tasks including measuring energy consumption of packet transmission, machine learning algorithms, and encryption algorithms.

1.2 Profiling and Improving the Duty-Cycling Performance of Linux-based IoT Devices

Among the most-popular operating systems (OS) for IoT edge devices is Linux. According to the Eclipse IoT survey report [37], 71% of IoT developers rely on this OS. Existing low-cost devices such as the Raspberry Pi 3 (RPi3), Raspberry Pi Zero Wireless (RPiZW), Arduino Yun, and Beaglebone Black support this OS. Despite low-level software and hardware improvements targeted at lowering their energy consumption, widespread use of these devices in IoT contexts requires employing user-level energy conservation techniques. Although there are valuable studies on the power measurement and modeling of this device [38, 39, 40, 41], unfortunately, less attention has been paid to improving energy efficiency.

One of the most effective approaches towards improving energy efficiency is *duty cycling*. This approach has been widely employed by the wireless sensor network community to achieve a long node lifetime, in some cases up to a few years [5, 26, 17]. In a duty cycled application, a device powers on, performs its intended task, and then powers off for a specified interval. Without relying on duty cycling, the lifetime of a system using an RPi3 and a 2400mAh battery is around 6 hours, assuming 400mA and 5V for current and voltage consumption.

The major burden of a duty-cycled Linux-based device is its boot up time. Specifically, the system software and hardware must be loaded and initialized before running user applications. Compared to real-time OSs such as FreeRTOS [15] and ThreadX [14], the boot up time of Linux is several orders of magnitude longer. To show this, we measured the boot up time of various hardware platforms when using Linux, FreeRTOS and ThreadX. The results are presented in Figure 1.1. These results indicate that for Linux-based systems a significant amount of

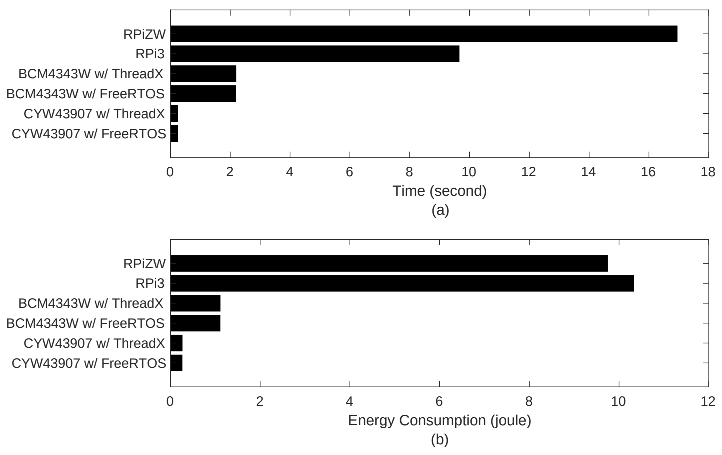


Figure 1.1: The (a) boot up time and (b) energy consumption of popular IoT operating systems on different hardware platforms.

energy is consumed during the boot up process at the beginning of each duty cycle.

There are two types of energy optimization techniques commonly applied to reduce the energy consumption of Linux: *general optimization* [42, 43, 44, 45, 46] and *application-specific optimization* [47, 48]. General optimization refers to the improvement of OS code to run faster and consume less energy, agnostic to the application type. Sample techniques belonging to this category include improving the filesystem and compression methods of unpacking the kernel during the bootloader phase [42], running boot up scripts in parallel and disabling kernel print statements [43], and saving a copy of the boot image to a file for reuse on subsequent boot ups [44]. Applying general optimization techniques, however, requires a deep understanding of the boot up process and OS; a trait that most developers may not posses. In addition, hardware initialization may pose some challenges.

For example, assume that a system image is taken after the hardware devices have been initialized. When this image is reloaded during the next duty cycle, some hardware devices might not be initialized, and therefore the user application cannot function properly. Addressing this limitation requires deep system knowledge and thorough testing to ensure system reliability. Additionally, applying security updates requires image regeneration. Even if the image can be updated, reliability would be a major concern. If images are generated on-the-fly after updates, the system might fail to boot up if the image is corrupted. Enabling fallback images requires an overhead for image verification and uses at least twice the storage space.

Application-specific optimization, on the other hand, refers to either OS improvement or the removal of unnecessary components, depending on application requirements. For example, by customizing the bootloader and thinning the kernel of unnecessary modules, the authors of [47] decreased the boot up time on an embedded Android device by 65%. In [48], the authors optimized a Linux-based smart television and reduced its boot up time by five seconds. However, since they defined boot up time as the interval between power on and the time instance at which the user can interact with the device, they delayed the initialization of certain components until after the home screen of the television was loaded. Unfortunately, the existing application-specific optimization techniques do not provide simple and universally applicable guidelines for IoT scenarios. When Linux-based devices are used in IoT applications, it is desirable to tailor the system based on application requirements. Even the Linux distributions released for IoT boards are preloaded with unnecessary services and initialize hardware devices not required by specific IoT application scenarios. For example, a device running an image classification algorithm may not require sound utilities, remote login, service discovery daemons, time synchronization, or any of the wireless technologies offered by the device.

In Chapter 4 we focus on the userspace level and application-specific optimization to improve the performance of duty-cycled Linux systems. The goal of this paper is to profile and improve the energy consumed by the boot up and shutdown phases of the RPi3 (Raspberry Pi 3, based on the quad-core BCM2837 SoC) and the RPiZW (Raspberry Pi Zero w/ Wireless, based on a single-core BCM2835 SoC) in such a way that minimal work is required to tailor a standardized installation image to specific IoT applications. In particular, the contributions of this paper are as follows: first, we overview the boot up process and present the implementation of a testbed to measure the duration and energy consumption of this process. In order to reveal the effect of loading services on system performance, we profile the start and end of loading units that require more than 10ms to initialize. By categorizing system units into multiple classes, we show that customizing the set of active units, which we refer to as *unit configuration,* is a very effective approach towards improving duty-cycling performance. For example, unit configuration reduces the energy consumption of the boot up process using a RPi3 by 43.62% for an application that only requires communication with the camera interface. In addition, through profiling the resource utilization of the RPi3 and RPiZW in terms of processor, memory and I/O, we further analyze the overhead of initializing units and show the possibility of running user application processes while userspace initialization is still in progress. Second, in the context of flash memory, even when up to 95% of its capacity is utilized, our results show no effect on the boot up duration or energy consumption. However, using faster flash memory always results in a slightly lower boot up duration (around 1.5%) and energy consumption (around 2.5%). Third, we investigate two shutdown approaches, *graceful* and *forced*, and evaluate the impact of unit configuration on their performance. Our results confirm that using unit configuration reduces the energy consumption of the graceful and forced shutdown by up to 43.87% and 57.42%, respectively. Additionally, the benefits and risks of both categories are highlighted when used for duty-cycled systems. Fourth, we propose *Pallex*, a parallel execution framework to execute a user application while userspace initialization is still in progress. Using Pallex, a user application is split into several stages that execute at different points of the userspace initialization phase. Our evaluations considering different user application scenarios show that in terms of lifetime, Pallex improves the duty-cycling performance of the RPi3 and RPiZW by 30.2% and 9%, respectively. Our results also confirm that although the power consumption of the RPi3 (quad-core) is higher than that of the RPiZW (single-core), the RPi3 achieves a longer lifetime due to its significantly shorter processing duration.

We have chosen to conduct our research using Raspbian Stretch Lite (RSL) on both the RPi3 and RPiZW because around 43% of Linux-based IoT systems rely on Raspbian [37]. RSL is a popular release packaged without a desktop environment, is advertised as a minimalist distribution, has a long development cycle, and is officially supported by the Raspberry Pi Foundation, thereby making it an ideal candidate for deploying energy-efficient IoT applications. However, it is worth noting that the results of this paper translate over to other Linux distributions supporting `systemd` such as Debian, ArchLinux, and Kali.

Chapter 2
Related Work

In this chapter we list and provide analysis on solutions that are either currently used or proposed by the research community. In Section 2.1 we describe current energy measurement solutions and explain their shortcomings. In Section 2.2 we explore current solutions used to improve the boot process on Linux based IoT devices.

2.1 Energy Measurement

In this section we provide an overview of existing and proposed solutions to measure energy on IoT devices. We categorize these based on their main shortcoming: (i) costly and bulky, (ii) complex circuit, (iii) limited supported range,and (iv) low accuracy.

2.1.1 Costly and Bulky

Most existing power measurement platforms are costly and bulky. For example, in [49] an oscilloscope is used to analyze the energy consumption of 802.11 tranceivers. A current probe is used in [50] for power monitoring. However since the probe relies on the magnetic field generated by current draw, it cannot be used to detect small currents or variations in the milliamp level. The authors in [19] added a shunt resistor to a USB cable and measured both current and bus voltage using the USB1608-FSPlus [51], a 16-bit analo digital converter (ADC). The USB1608-FSPlus is controlled by a PC and the sampling rate is 1Ksps. All these approaches are expensive, moreover their size prevents them from integrating with an IoT testbed.

2.1.2 Complex Circuit

Similar to integrated circuits (ICs) such as BQ2019 [52], the SPOT [32] platform uses voltage to frequency conversion and relies on the resources of the device under test to operate. In Cdition, the oscillator and converter introduce noise and error which may interfere with the device under test. SPOT also is not plug-and-play. While SPOT's measurement error is less than 15% with 1μA resolution it also assumes the typical operating range of the IoT device to be between 5μA and 50mA. LEAP2 [24] is a FPGA-based platform that enables individual monitoring of various components such as processor and memory. iCount [33] relies on the linear relationship between current and switching frequency of boost switching regulators: counting the switching cycles reflects the current drawn during the interval. However, the main shortcoming of iCount is that it uses the resources of the device under test which introduces extra overhead that contributes to increased energy consumption. Some complex circuit solutions may require extra hardware such as boost converters which are not always available on IoT boards. Other solutions such as CYW43907 [11] and CC2650 [53] use internal voltage regulators. This significantly limits their range of support for various IoT devices.

Energy Bucket [31] counts the number of charges and discharges of a buffer capacitor. One of its advantages is that it can measure currents in the range 1μA to 100mA. However, because of this approach Energy Bucket's sampling rate is fully dependent on the buffer capacitor value. In other words, a smaller current draw will result in a longer inter-sampling interval. In addition, the platform assumes the bus voltage is fixed and no accuracy analysis is provided.

μMonitor [54] is a platform which proposes a power monitoring platform based on counting capacitor charge and discharge cycles. For loads within 1μW and 10mW, the accuracy of μMonitor is almost within 10% of the results obtained from a 16-bit ADC that digitizes the voltage value over a shunt resistor. Unfortunately, the evaluations performed use static loads. Similar to Nemo [21], Potsch et al [55, 56] propose the use of two shunt resistors (1Ω and 100Ω) for measuring low and high currents up to 100mA. Shunt resistor voltages are amplified and then sampled by a 16-bit ADC which is then collected by a 32-bit microcontroller. We do not know the actual sampling rate and resolution as neither of those parameters have been evaluated.

The complex circuitry of these platforms is another shortcoming of the proposed platforms. Each of them include various complex components such as ADC, op-amp, resistors, high precision capacitors, and extra processors. This complex circuitry makes it more difficult, costly, and time consuming to build. On the other hand, EMPIOT is very easy to build and the cost of a complete platform including the base board is $35 when using a RPi3 or $15 for a Raspberry Pi Zero with WiFi (RPiZW).

2.1.3 Limited Range

PowerBench [30] is capable of providing 5KHz sampling rate and 30μA resolution. It assumes the maximum current consumption of the host device is 65mA. This platform samples the amplified voltage (51x) across a shunt resistor using a 12-bit ADC. In [22], the authors assume energy profiling requires only current measurement which limits functionality of the described platform. PowerBench's error can be lesser than 5%, but only as long as current is higher than 20μA. However, the maximum supported current is 35mA. iWEEP_HW [57] employs a multi-layer architecture to measure the power consumption of processor, tranceiver, and sensors using separate ADC channels. The platform uses a PIC18 processor and 10-bit ADC which measures voltage across shunt resistors. iWEEP_HW can measure currents up to 40mA with a maximum sampling rate 150KHz. PotatoScope [58] is a microcontroller-based oscilloscope that focuses on reliable energy measurement in outdoor environments especially in those with significant temperature variation. PotatoScope includes an ARM Cortex-M3 processor attached to a 12-bit ADC to sample current and voltage. The voltage across a 0.47Ω shunt resistor is amplified by a factor of 200. Since the ADC's reference voltage is 2.5V, the maximum measurable current is 26.6mA. The main disadvantage of using these platforms is their limited current measurement range which is less than 100mA. IoT devices rely on high data rate technologies which increase power consumption as high as 700mA. Therefore none of these platforms can be used with IoT devices. On the other hand, the performance evaluation results presented in Section 3.3 confirm EMPIOT's effectiveness for measuring energy in 802.11 devices.

2.1.4 Low Accuracy

Energino [20] is a platform which uses the 10-bit ADC of Arduino boards. The ADC supports input voltage 0 to 5V, therefore the LSB is 4.88mV and the current measurement resolution is 25mA. Energino uses a Hall-effect current sensor with sensitivity 185mV/A to which attains a sampling rate of 1KHz while measuring currents up to 5A. NITOS [59] uses the ATmega2560 micro-controller with a 10-bit ADC. Since the Arduino's ADC cannot measure millivolt level variations in voltage, it is required to use a voltage amplifier. In order to maximize sampling rate, NITOS uses the ADC in free running mode which increases the ADC's prescalar clock from 125KHz to 1MHz, and uses interrupt service routine to get ADC values. Although these enhancements result in up to a sampling rate of 63KHz, the accuracy of ADC is reduced by 11% due to the higher clock being used. Furthermore, the current measurement of the platform is 25mA.

2.2 Optimizing Linux Boot Process

In this section, we compare existing solutions relevant to userspace initialization optimization. For the sake of completeness, we also study several other techniques for improving Linux boot up, namely, bootloader optimization, kernel space optimization, and suspend optimization. Variants of these techniques can be utilized in conjunction with our work to further enhance boot up time and energy consumption.

2.2.1 Userspace Optimization

The authors in [45] identify several areas of improvement for boot up time on Debian, a Linux operating system, which at the time used `sysvinit`. Although Raspbian uses `systemd` instead of `sysvinit`, its backwards compatibility allows for some of these optimizations to translate over to devices using `systemd` as their `init` system. The first area of optimization involves substituting lighter default applications. For example, boot up time was improved by two seconds by using `dash` instead of `bash`, the default shell for Debian. Rewriting slow shell scripts to use internal functions instead of external functions has also been proven to improve the performance of boot up time.

Boot scripts can also be executed in parallel. This is achieved through setting the `CONCURRENCY` option in each boot script. Reordering boot scripts can also help programs complete faster. In [48], the authors identified areas during the boot up process of a smart TV when processor utilization is low. They used this information to optimize boot script execution by interleaving I/O-intensive programs with processor-intensive programs. While this can be an effective solution, maintaining system stability can be troublesome, as `sysvinit` does not track dependencies. Therefore, it is the user's responsibility to ensure any changes do not compromise system stability. Compounding the issue, dependencies between boot scripts are often not well documented and require thorough research and an understanding of both the Linux kernel and `sysvinit` before modification is possible. By comparison, `systemd`, a modern `init` system, tracks dependencies and automatically generates a tree structure to determine which startup tasks can be executed in parallel. Therefore, the feature set of `systemd` makes boot script reordering a much simpler task.

In addition to improving boot up time, we studied several methods for shortening the duration until *user interaction time*: the time it takes until the user is able to interact with the application. In [45] the authors are able to gain an improvement of two seconds by performing certain tasks, such as setting up the network and hardware clock, in the background. In [47, 48], the authors optimize boot up time on Android smartphones and televisions by deferring long-running services until after the log-in screen is displayed to the user. However, this approach is not directly applicable to the IoT devices employing duty cycling.

Application XIP (Application Execute in Place) [46] is another technique for improving application start up speed. *Application XIP* allows for user applications to be executed directly from the filesystem instead of first being loaded into RAM. When a program is executed, the kernel program loader maps text segments for applications directly, resulting in a reduction in RAM footprint, faster first invocation, and reduced power consumption of flash vs RAM. However, *Application XIP* requires compiling the kernel with support for a filesystem (such as CRAMFS [60]) that supports storing files in contiguous blocks of memory. Additionally, there are hardware requirements for XIP such as random access storage that is directly accessible by the processor, which the RPi platform does not offer.

One major complexity of userspace initialization can be attributed to the evolution of the `init` system's responsibilities. Modern `init` systems often manage process scheduling, I/O scheduling, and memory scheduling, among other responsibilities. In [61], the authors addressed this issue by simplifying the existing `init` scheme to improve an Android device's boot up time. Their contribution involves separating the `init` scheme into two booting modes: normal boot, which executes all tasks during boot up time, and quick boot, which executes only mandatory tasks before user interaction time. However, duty-cycled IoT systems do not typically require separate modes for mandatory services and extra services, as an optimized system only runs the software required to complete its task. When device features, such as applying system updates, are not utilized during every duty-cycle, they are easily loaded after the boot up is completed. Furthermore, modern Linux kernels support dynamic kernel module loading using the `modprobe` utility.

2.2.2 Suspend-related Improvements

Another approach to improve boot up time is the usage of a suspend or hibernation mechanism. In [62], the standard *hibernate-resume* approach is used to optimize the boot up time of a digital camera. During the *system suspend* phase, current system information such as processor registers, I/O map information, and runtime variables (both global and local), are stored in RAM before the device powers off. When the system receives certain power events, the *resume* operation is started. The previous state of the device is restored from RAM, resulting in a faster initialization time than a regular boot up. It is important to note that the RPi platform's hardware does not support hibernation or suspension, as the SoCs do not support modern power management features [63].

In [44] and [64], methods for *snapshot boot techniques* are discussed with respect to boot up time improvement on a device running Linux. These methods utilize a snapshot image created at boot up time and stored on a disk or in reserved flash memory. For subsequent boot ups, the device loads the suspended image instead of stepping through the standard kernel initialization process. As opposed to a standard *hibernate-resume* operation, this snapshot image is generated only once and then reused. A major disadvantage of snapshot images is the considerable amount time required for image generation, verification, and storage. Also, storing a full image in addition to the original system scripts and binaries necessary for image generation requires a considerable amount of disk space, which may not be available on many IoT platforms. If the image is corrupted (due to power loss, for examples), the device may become unrecoverable if a fallback image is unavailable.

2.2.3 Bootloader Optimization

The bootloader is responsible for initializing the hardware and loading the kernel. In most cases, the kernel is compressed to save space and the bootloader must decompress it before use. The authors in [42] analyzed the time required by the bootloader and compared the performance of several root filesystems for fast boot up time. Decompressing the kernel with `gzip-cheksum` on their system resulted in a bootloader time of 16s. Conversely, using `gzip-nochecksum` required only 12s, with a reduction of 3.8s. While the actual decompression of the image took 2.79s, verifying the checksum required an extra second on their device. In addition, the cost of calculating the checksum for the RAM disk was 2.78s. They found that storing the image in a decompressed format can circumvent this process. These results are noteworthy in the realm of embedded Linux, where processors may be slower than the RPi's processor and the onboard flash is more responsive than the SDC used by the RPi. However, most general-purpose Linux systems can decompress the kernel faster than the storage medium can read the uncompressed alternative. Therefore, it is important to balance the speed of the processor and the I/O read speed before seeking performance gains from an uncompressed kernel.

Similar to *Application XIP*, *Kernel XIP* (Kernel Execute-In-Place) [46] is a popular technique used for optimizing bootloader initialization. In a typical boot sequence, the kernel is decompressed either during or just after it is loaded into memory. XIP enables direct execution of kernel instructions directly from ROM or flash memory. However, this method requires the kernel be stored in an uncompressed format, thereby requiring more storage space. The RPi platform does not support `Kernel XIP` due to hardware constraints.

2.2.4 Kernel Space Optimization

In [42], the authors show the effect of removing unnecessary kernel modules and bundling multiple modules into a single module on decreasing kernel loading time. In another work [46], the authors evaluate the effect of disabling kernel print statements to prevent bottlenecks caused by streaming messages to the console. The `quiet` option in the kernel configuration changes the logging level of print statements to 4, which suppresses the output of regular (non-emergency) messages.

Chapter 3

An Energy Measurement Platform for Internet of Things Devices

In this chapter we present EMPIOT, a low-cost and effective energy measurment platform for IoT devices. In Section 3.1 we describe EMPIOT's hardware, software, and features. In Section 3.2 we describe ProCal, the calibration tool used to calibrate EMPIOT for individual devices. In Section 3.3 we present our findings on EMPIOT's performance on different IoT devices and discuss how various settings and parameters impacted its performance.

3.1 Platform Design

In this section we explain the hardware and software that makes up EMPIOT. We then mention key design parameters and study their effect on EMPIOT's performance. Finally we explain the motivation behind calibration for individual IoT devices and explain the calibration tool used to achieve this.

3.1.1 Hardware

EMPIOT 3.1 consists of a shield that is mounted on top of the GPIO pins of a base board. In our case it was mounted on a Raspberry Pi 3 (RPi). The main

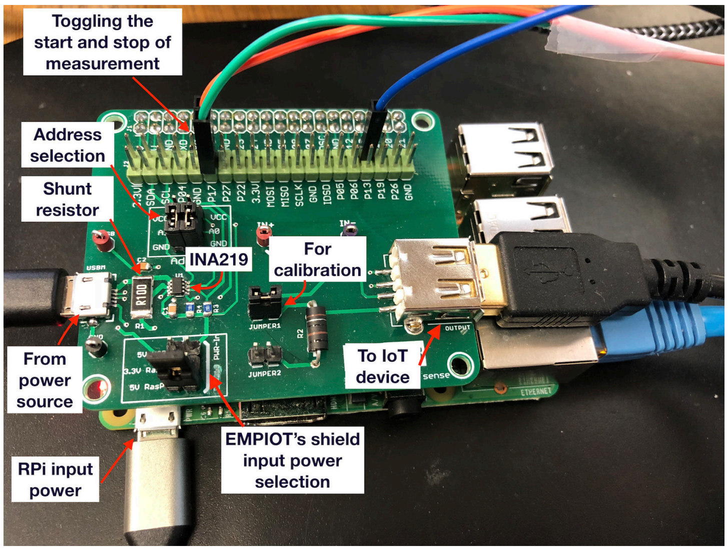


Figure 3.1: The EMPIOT shield mounted on a Raspberry Pi 3. GPIO pins are connected to the board under test.

component powering the shield is the INA219 [34], an inexpensive ($2) analog digital converter (ADC) which provides readings for current, voltage, and power. The INA219 chip comes in two versions - INA219A and INA219B. We use INA219B because it has lower (0.5%) variations versus temperature. The energy draw of the chip is 1mA and it can operate using a 3.3 or 5V supply. INA219 measures bus voltage directly and current is measured through digitizing the voltage across a shunt resistor. The ADC's basic resolution is 12 bits, and 1 LSB step size for shunt voltage and bus voltage are 10μV and 4mV, respectively. The ADC is a delta-sigma type and uses a frequency of 500KHz to collect analog samples. After collecting samples, the delta-sigma ADC uses a low pass digital filter to reduce noise and a decimator to average the analog samples. According to the INA219 datasheet, the duration of these operations are 532-586μ for 12 bit resolution and 84-93μ for 9 bit resolution. However, we will show in Section 3.3 that the actual sample preparation is longer than these reported values. The full scale voltage range supported across the shunt resistor is 40mV. For a 12 bit resolution, the shunt voltage is $40/(2^{12}-1) = 9.768\mu$V. For resistance, we used a 0.1Ω shunt resistor with 0.5% accuracy. Therefore, the current resolution for EMPIOT is 100μA for currents up to 400mA. Depending on the maximum possible current, the power gain amplifier (PGA) can be configured to achieve full-scale range through dividing shunt voltage by 2, 4, or 8. This means we can measure shunt voltage in the following ranges: [0, 40]mV, [0, 80]mV, [0, 160]mV, [0,320]mV. Depending on the applied configuration, the maximum supported bus voltage is either 16V or 32V. When configured for 16V, the accuracy of bus voltage measurement is $16/(2^{12}-1) = 3.907mV$. We assume current and voltage are less than 800mA and 5.5V since that is the nominal operational range of IoT devices. INA219 uses the I2C bus to communicate to a master device. The I2C bus supports a minimum speed of 0.1MHz and a maximum of 2.5MHZ. We decided to use an RPi as the underlying platform to configure and collect results from the chip because (i) it has a fast enough I2C rate to support INA219's generation rate, (ii) the memory card (and ability to expand storage) can be used for extended duration of power sampling, (iii) it can be used to communicate and activated by the attached IoT device through the GPIO pins or a socket, (iv) it comes with multiple communications (Ethernet Wifi, Serial) which simplify testbed setup and remote access.

3.1.2 Software

The EMPIOT shield relies on I2C communication to send data to the RPi. We decided to use and study the differences between two libraries - the Linux I2C library and the BCM2835 library. The BCM2835 library provides GPIO and other IO functions on the Broadcom BCM2835 chip, as used by the RPi, allowing access to the GPIO pins. In addition, it provides functions for reading digital inputs and setting digital outputs using SPI and I2C. The Linux dev library is another library which is native to the Linux kernel. Similar to the BCM library, it provides a library for I2C, SPI and other IO interfaces. We chose to use both of these libraries because they were written in C (which integrated with our C++ program) and because they were the two most popular, having support and documentation from their respective communities.

Algorithm 46 shows the pseudo-code for the software the base board uses for collecting and saving data. At a high level, the program consists of the `main` function and two threads. The `main` function is responsible for initializing the I2C driver, shield, and the two threads. It first writes to the INA configuration registers to adjust gain and resolution. The gain determines the maximum measurable current and the resolution refers to the number of bits per sample. It then initializes the I2C driver and configures the bus speed which determines our sampling rate. Finally, the `main` function creates the `sampler_thread` and the `file_writer` thread.

The `sampler` thread is responsible for interfacing with the INA chip. It polls the chip's conversion ready bit and when set, reads the the bus and shunt voltage values. To calculate current, we multiply the bus voltage by 10 which is the resistance of the INA chip. While we could have read the current register on the INA chip, we decided to instead move that calculation to the software since reading from an extra register takes more time. If energy saving mode is enabled, the `sampler` thread calls the `compute_energy` function which calculates energy consumption via the Riemann integral approach. After collecting the data, the `sampler` inserts the new entry to a buffer data structure which is responsible for flushed to a text file. We implemented and experimented with two different data structures for storing the data in memory, a two-buffered and a circular approach.

The unique nature of both these data structures also enable us to experiment with different methods of file writing. The two buffered algorithm uses two different fixed-size arrays which are allocated with a set size when the program starts. These arrays are declared as fixed size and only initialized once at the beginning of the program using stack memory to optimize performance. When the buffer in use reaches maximum capacity, we swap it with the unused buffer and save the used buffer to file. The two buffer approach supports *batch writes* where entries are saved every time a buffer gets swapped. We can configure the batch size by changing the size of the buffers which in turn impacts both performance and energy consumption. The circular buffered data structure only requires one contiguous array. New entries are inserted to the beginning of the circular buffer and entries at the end are save to file. Because one array is being accessed by both threads, a `mutex` is used to dictate access. When the buffer reaches maximum capacity it stops accepting new entries until space has been made available. Due to its nature, the circular buffer supports *continuous file writes*.

The EMPIOT software supports both raw data collection and energy measurement. Raw data collection will take readings of a timestamp, shunt voltage, bus voltage, and shunt current. Energy will take readings of a start timestamp, an end timestamp, and the energy consumed during that duration in joules and nanojoules. The program can run in three different modes: (i) a given time duration, (ii) a number of samples collected, and (iii) a `trigger` mode which allows the system to be notified when to start and stop sampling. This trigger can be activated either through the GPIO pins or by sending a packet through a socket on port 5000. This feature allows the Pi to be controlled by the board under test, remotely from another device, or even by another program running on the Pi. In addition, EMPIOT supports both 9 and 12-bit resolution samples and 400 and

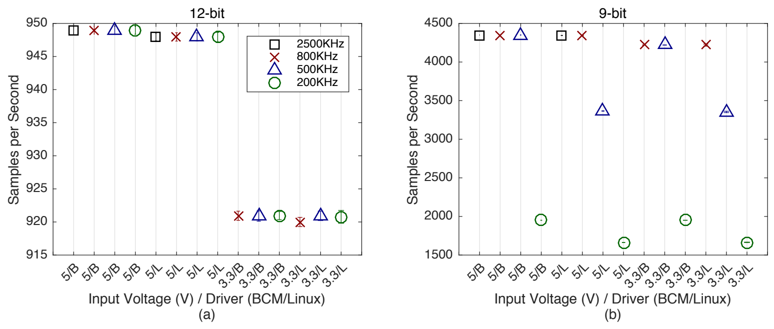


Figure 3.2: Sampling rate for (a) 12-bit resolution and (b) 9-bit resolution. Error bars show 95% confidence interval.

800mA calibration.

3.1.3 Design Parameters

In order to achieve maximum performance, we analyze the effect of various design parameters such as sampling rate, sampling offset, accuracy, and energy consumption of the base board.

Effect of the input voltage, driver, and bus speed on sampling rate

Figure 3.2 shows sampling rate with respect to bus speed, driver, and input voltage. Input voltage refers to the power source of the INA219 on EMPIOT's shield.

We observe the following: First the sampling rate is lower than the conversion rate supported by the chip. For example, for 12-bit conversion, the sampling rate is lower than the 1.8KHz ADC conversion rate as reported in the INA219 datasheet. A second observation is that a lower voltage does result in a lower sampling rate. We also notice that reducing the bus speed to as low as 200KHz results in a decrease in sampling rate. Finally we observe that the Linux driver affects sampling rate for 9-bit resolution.

In order to evaluate software overhead, we measured the time spent by the `sampler` thread between collecting a sample and the next polling of conversion ready bit. Using a high-speed logic analyzer, our results show that the processing overhead of the `sampler` thread to be a negligible 0.46μs. In order to measure *I2C read delay*, we created a program that continuously reads two bytes from the EMPIOT's shield board. After capturing I2C traffic by the analyzer we compute the interval between sending I2C addresses. Figure 3.5 shows the results[1] which reveal the effect of bus speed and driver on read delay. In particular, the BCM driver achieves a faster and more stable I2C performance compared the Linux driver. More specifically, for any given baud rate Linux I2C read speed is at least 20μs slower than that of BCM. To more empirically justify the lower performance of Linux driver with respect to BCM, we used the `strace` [65] program to log system calls made by the software. Our experiments show that when using BCM, the number of system calls is always fixed (exactly 181) and does not depend on the sampling rate. We also observed that these system calls are only made during initialization of the BCM driver as communication over I2C occurs directly with the driver rather than going through the kernel. On the other hand, all I2C communication requests using the Linux driver pass through the kernel; which results in a linear increase between system calls and sampling rate. Figure 3.5 also shows that the I2C delay of Linux driver has higher variations compared to BCM: the average range of variations for Linux I2C read delay is 22μs, while the rate for BCM drops to 4μs. These results indicate that the Linux driver does not achieve a reliable rate communication with mission-critical and high rate sensors, such as those used in medical and industrial applications. The effect is also obvious when using 9-bit sampling (cf. Figure 3.2(b)). For example, when using a bus speed

[1]We did not report the results for bus speed 2500KHz and input voltage 3.3V because we observed a very unreliable I2C communication in this condition. We believe that INA219 cannot keep up with this high clock rate when the voltage is 3.3V

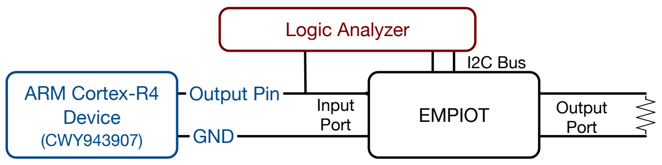


Figure 3.3: In order to measure I2C read delay, the logic analyzer captures I2C communications. To measure sampling delay offset, the logic analyzer captures both the status of the ”Output Pin” and communication over I2C.

of 500KHz, using the Linux driver reduces sampling rate to 3360, compared to the 4350 samples collected per second when the BCM driver is in use. From the software point of view, the overhead of Linux driver affects the number of times the conversion ready bit is polled per sample collection round. Table 3.1 reports the results. Therefore we conclude that the Linux driver falls behind the sample conversion rate when the sampling rate is high.

Next we analyzed the time interval between a change in input and reading the corresponding value which we call the *sampling offset*. In order to identify the causes of sampling offset, we used the experiment shown in Figure 3.3. To introduce a quick and predictable change in power, we have used a pin toggle using an ARM-Cortex R4 board (CWY943907 [11]) which sets a pin from high to low (i.e., 3.3V to 0V) in 50ns. For this experiment, the ”Output Pin” in Figure 3.3 is initially high (3.3V). At particular intervals, the pin is set to low (0V) and remains in this state for 2ms. We compute sampling delay offset through measuring the interval between setting a pin to low and collecting the corresponding sample. To this end, we use a logic analyzer to log the status of ”output pin” as well as communications through I2C.

Figure 3.4 shows the measured values. Since I2C read delay is independent of input voltage (as Figure 3.5 shows), the results of Figure 3.4 indicate longer conversion time when using a lower voltage value, i.e., 3.3V. For example, increas-

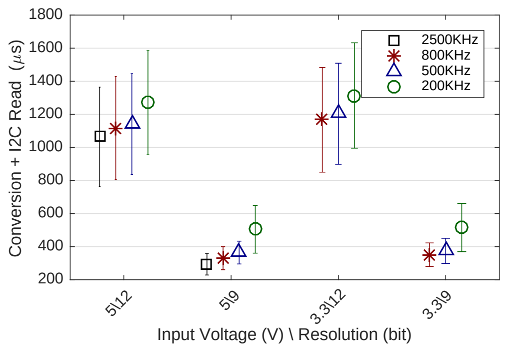


Figure 3.4: Sampling offset, which reflects the interval between a change in input current and reading the corresponding value. For this experiment we used the BCM driver. Error bars show 95% confidence interval.

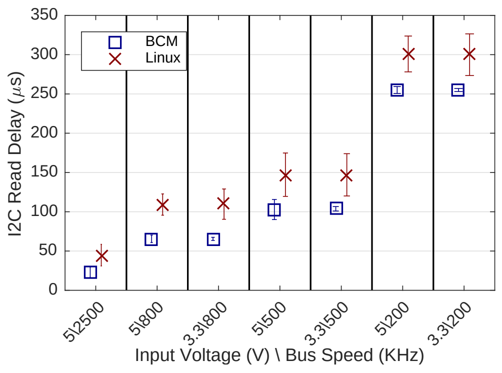


Figure 3.5: I2C read delay measured as the time interval between issuing a read command and reception of requested bytes. Error bars show 95% confidence interval.

ing conversion time by 64μs decreases the number of samples collected per second by 47. As the power measurement chip uses a delta-sigma ADC, we believe that the lower voltage value slows down the operation of the decimator and averaging circuitry.

Effect of Input Voltage on Measurement Accuracy

In this section we analyze the effect of input voltage variations on accuracy. We tested EMPIOT's shield board using the following voltage sources.

– External (xVExt): A xV external DC power supply [28]

– 5V from RPi (5VRPi): The 5V pin of RPi GPIO pin.

– 5V from RPi with a load (5VRPi w/Load): The 5V pin of RPi GPIO pin. To generate a load, we connect RPi's USB port to a Cypress CYW943907 IoT device. The device's idle energy consumption is about 100mA, but periodically wakes up and sends ping packets that result in up to 400mA current consumption.

– 3.3V from RPi (3.3VRPi): The 3.3V pin of RPi GPIO pin.

To determine our ground truth we decided to use an industrial-grade DMM [29] in order to accurately measure variations of these power sources. The results are as follows: 5VExt: 2mV, 5VRPi: 103mV, 5VRPi w/Load: 300mV, 3.3vRPi: 24mV. We observe that using RPi as a power source exhibits higher variations compared to 5VExt which we attribute to background running processes from the operating system. In spite of this, we still want to confirm if RPi can be used as a source of power as being able to use it without any degradation in performance can greatly simplify the platform's design.

In order to measure accuracy across a wide range of variations, we used three different fixed loads: 200μA, 5mA, and 100mA. These loads are generated by connecting the shield's output to three different resistors. Figure 3.6 presents the effect of voltage variation on the accuracy of bus voltage and current measurement. These results show the variability of measurements caused by factors such as electromagnetic interference and white noise. In spite of this we observe that irrespective to the source of power and load values, the measurement error of EMPIOT is always less than 4mV and 0.1mA respectively for bus voltage and current.

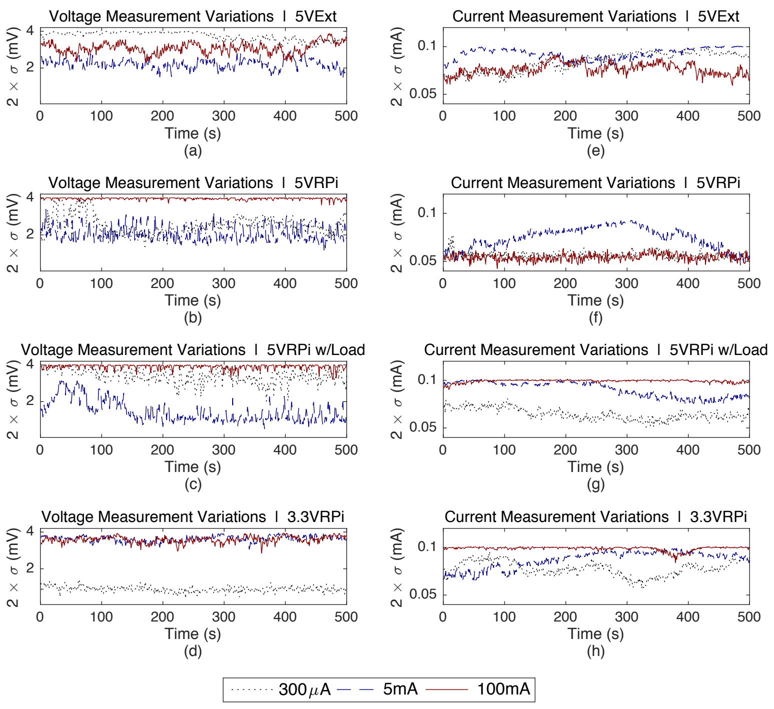


Figure 3.6: The measurement variations of voltage (sub-figure (a) through (d)) and current (sub-figure (e) through (h)) over time for three different loads and various input sources. Sampling resolution is 12 bit. The y-axis is the 95% range of variations.

The reported error ranges comply with the values we mentioned early for INA219 in (cf. Section 3.1.1).

Effect on buffering mechanism, driver, and sampling rate on energy consumption

We investigate the energy consumption of the RPi running EMPIOT's software. In order to do this we used two EMPIOT boards where one measures the energy consumption of the other. The board under test takes measurements of an IoT board in idle mode. In addition to energy, we have logged the time we write each sample to file by toggling a pin and logging pin activation times. By doing this we are able to avoid introducing the extra software overhead necessary for logging file writes. To remove variations caused by Ethernet communication, we used UART to communicate with the RPi board under test. When using UART instead of Ethernet we notice the energy consumption of the RPi to be reduced by 30mA in addition to reducing power variations.

Figure 3.7 shows the results of both approaches. The red trace displays the energy consumption of the RPi that was measuring the IoT board. Blue bars indicate the time the RPi wrote each sample to file. Using the circular buffer results in continuously writing to the flash memory which in turn increases energy consumption. The energy consumption stays in a relatively fixed range and we observe very few prolonged spikes in power. For example, for 9-bit resolution and BCM driver, the base power of the RPi is increased from 1.25W to 1.3W when using the circular buffer mechanism (compare subfigure (a) and (c)). In this case, the circular buffer increases the activity time of the `sample_writer` thread and increases processor utilization by about 15%. Since the `sample_writer's` processor core utilization is around 98%, this increase requires the utilization of another processor core, thereby preventing core sleep.

Figure 3.7 also reflects the higher energy consumption of the Linux I2C driver. For example, when using 12-bit resolution, using the Linux driver increases the base power to 1.38W, compared to 1.26W achieved with the BCM driver (compare sub-figure(e) and (f)). As discussed earlier, I2C communication through the Linux driver results in a significantly higher number of system calls, which increases processing load.

Even though we use the RPi3 as EMPIOT's base board due to the benefits of having an Ethernet port and a multi-core CPU, these are not required and EMPIOT can still perform well on cheaper more low powered and less expensive boards. Our studies show that for a RPiZW with an active Wi-Fi connection, using the BCM driver and batched write operations results in 670mW power consumption. On the other hand using the circular buffer and Linux driver increases power consumption to 718mW and 685mW respectively.

Although we have used a small buffer size (20,000) for these experiments to reveal the effect of batched write operations, a much larger buffer size results in a higher energy saving. In our implementation, each sample entry is 16 bytes: 8 bytes to store in `timespec`, 4 bytes for bus voltage, and 4 bytes for current. Even if only 30% of the 1 GB RAM of RPi3 or 25% of the 512MB of RPiZW is used, then the number of entries kept in random access memory before each file write will be 19,660,000 and 8,388,608, respectively. We conclude that batching file writes through the two buffered approach can be easily used to minimize energy consumption.

3.2 Calibration

Various factors such as the resistance of the shunt resistor path, inaccuracy of shunt resistor, and ADC non-linearity are responsible for differences in EMPIOT's measurements and a ground truth. For these reasons calibration is important for EMPIOT's design as a well-calibrated system make up for these discrepancies.

In order to perform current calibration, we need a load that can generate currents within the supported measurement range of EMPIOT. Although current variations can be generated through an IoT device, this method does not result in an accurate calibration for the following reasons [19, 20, 21, 22, 23]. First, the duration of a change in current draw may not be long enough to match the samples collected. Due to fast variations of current and difference in sampling offset of the DMM and EMPIOT, correlating the samples collected by both devices is a

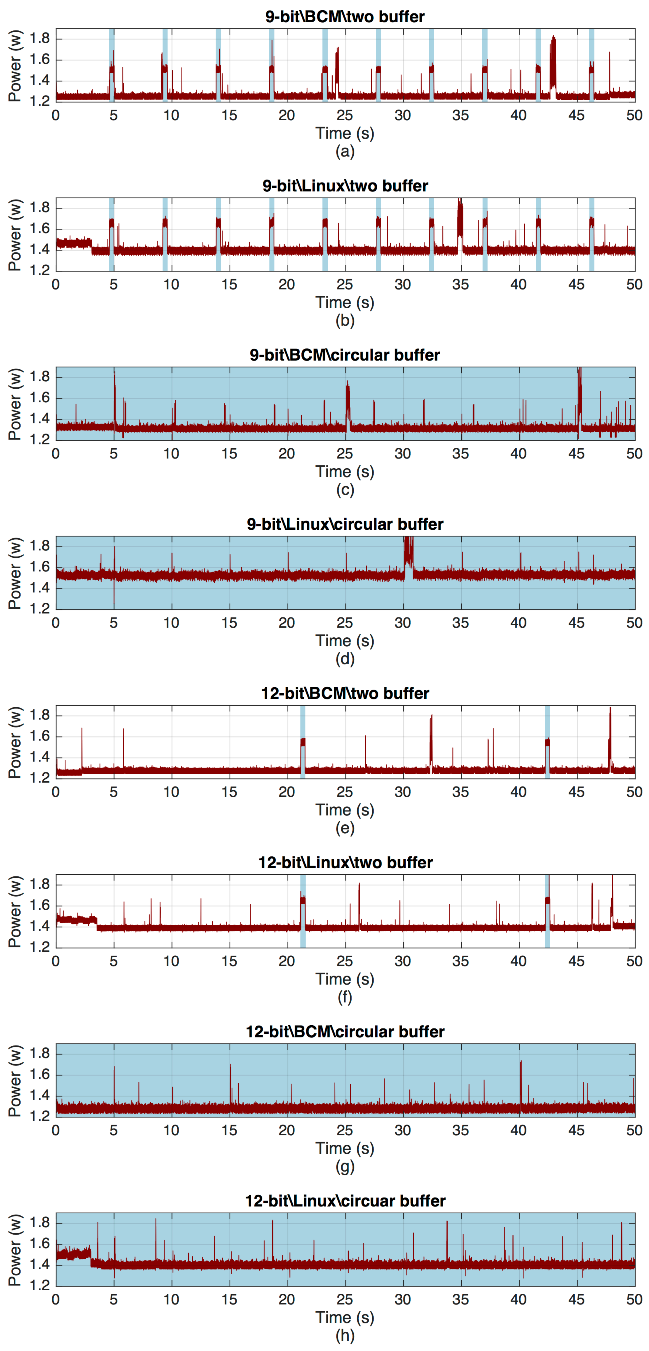


Figure 3.7: The effect of sampling resolution, driver, and buffering mechanisms, on the energy consumption of the RPi running EMPIOT software.

challenging process. For example, when the current draw changes suddenly, some values may not be captured by EMPIOT due to its lower sampling rate compared to that of a DMM. Furthermore, DMM and EMPIOT might report the variations with different time offsets which means that subtracting the pairwise values does not reflect a realistic measurement error. If the error of EMPIOT versus DMM is in fact a linear function, then these calibration errors previously mentioned may prevent us from finding and at best, cause a linear function with a slope which deviates from the real value. In addition to this, the IoT board may not cover the supported current range of EMPIOT which results in calibration gaps. In fact, turning on each component on the board results in a jump in energy consumption and is almost impossible to generate a linear increase. Another solution would be to use a potentiometer as the load. However, the drawbacks are similar to using an IoT device as we cannot predict the transition and duration of drawing a particular current value which results in the measurement offsets affecting calibration error. Furthermore, potentiometers usually support low current values and while calibration for currents higher than 100mA is possible, it requires an expensive potentiometer. Due to these limitations, the existing solutions perform calibration either (i) manually by using fixed resistor values [30, 33, 22] or (ii) using expensive equipment [66, 55, 54].

In order to create a scalable platform infrastructure for ADC calibration, we have designed a low-cost, accurate, and *programmable calibration tool*, named PCT which provides dynamic voltage and current ranges. With PCT, users can program a load and control its resistance and timing characteristics. A Python program running on a RPi is responsible for configuring PCT. This software controls reconfiguration frequency and records output settle times between two conservative configurations. More specifically, if reconfiguration frequency is t, the PFT software records the output settle times at $n * t + t/2$, where $n = 0, 1, 2, ...$ Recording output instances enables us to correlate the measurement values of DMM and EMPIOT when the load is stable. Furthermore, this feature prevents the need for accurate time synchronization of DMM and EMPIOT.

One of the key advantages of PCT is its large dynamic output range for both current and voltage. By using a high-accuracy digital potentiometer and well-calculated resistor combinations, the output range of PCT spans between 0.5mA to 1A for current and 0.06mV to 5.5V for voltage. We use 256-position 10k digital potentiometer ADS5200 [67] as the base block, and connect it in parallel with multiple resistor paths. Specifically, a new resistor path is enabled whenever a 20mA or a 100mA increase in current is required, and the digital potentiometer is used to fine-tune current in range 0-20mA. By programming the resistance value, PCT can generate various current and voltage values. The PCT software programs AD5200 and ADG1612 through SPI interface and GPIOs. Like EMPIOT, PCT is relatively inespensive as the total cost is about $100, including manufacturing costs which is less than 1% of the cost compared to current commercial solutions.

PCT generates line interrupts to trigger the start and stop measurements by EMPIOT and DMM. As mentioned earlier, EMPIOT supports measurement start and stop through line interrupts. To trigger the DMM we have used a high-accuracy and programmable device [29] that exposes several programmable GPIO pins. We have developed a Python script using SCPI (Standard Commands for Programmable Instruments) to configure the sampling rate and enable the DMM to start and stop sampling based on the line interrupts received. This script communicates with the DMM through a TCP/IP connection and samples the sampled data from the DMM buffer to a PC when a measurement completes.

EMPIOT's shield includes a jumper to enable current flow from input to output (cf. Figure 3.1). For calibration purposes we have removed this jumper and connected the pins to a DMM which allows both EMPIOT and DMM to measure current simultaneously. Instead of pairwise comparison of the traces collected by

EMPIOT and DMM, we use the timing data logged by the programmable load. As the timing data reflects load stability instances, we can safely compare the two closest entries of the traces collected by DMM and EMPIOT.

One of the goals of this paper is to show if an off-the-shelf INA219 breakout board can be used instead of EMPIOT's shield with the software and configuration parameters proposed in this work to achieve a high level of accuracy. In addition to reporting calibration data for EMPIOT's shield, we have also used a stock INA219 breakout board [68] to study the effect of hardware design on calibration. This breakout board has been installed on a bread board and communicates with a RPi running EMPIOT's software. Figure 3.8 shows the measurement errors of three EMPIOT boards and three breakout boards conducted in a normal indoor temperature $25°C$. Please note that the left value above each figure refers to the maximum supported current configured through the programmable gain amplifier (PGA). As it can be observed, EMPIOT's shield presents lower error compared to the breakout boards. Since INA219 measures current through a shunt resistor, the distance and impedance of a circuit path between the resistor and chip highly affect measurement accuracy. In addition, Figure 3.8 shows that the error of EMPIOT's shield increases linearly versus current. However, the breakout boards' error show a quadratic behavior for currents beyond 300mA. For both these cases, instead of using a calibration table, we simply find the best fitted curves. Table 3.2 reports the calibration values for these boards.

In addition to current, the voltage measured may not reflect the actual bus voltage. We used PCT to generate a variable voltage in the range 2V to 5.5V. Our results show that the error of voltage measurement is a fixed offset. In fact, EMPIOT's voltage measurements versus DMM results in linear function $V_E = V_{DMM} - 0.027$. For the breakout board the calibration function is $V_E = V_{DMM} -$

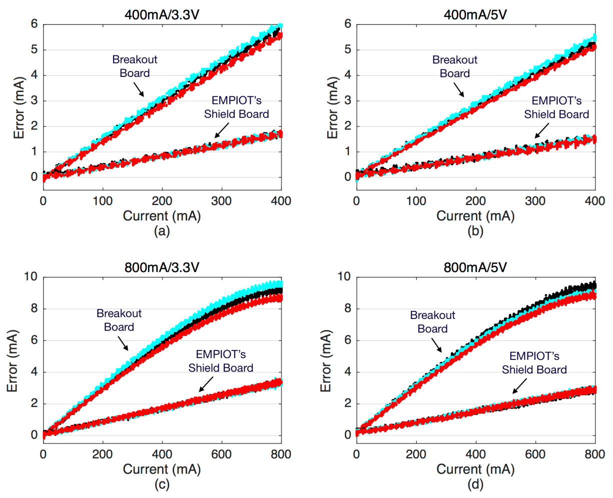


Figure 3.8: Current measurement error $(i_r - i_e)$ versus ground truth. The value above each figure shows the maximum supported current (left side) and input voltage (right side).

0.097.

3.3 Performance Evaluation

In this section, we study the effectiveness of design parameters and calibration on accuracy and energy efficiency when EMPIOT's shield and breakout board are used. The following settings are used: BCM driver is used, bus speed is 2500KHz, INA219's voltage is 5V, file writes are batched. Our ground truth is the energy measured by two high accuracy DMMs that record current and voltage with 500Ksps sampling rate and 18-bit resolution. A Python script is used to trigger the DMM in a similar way EMPIOT is triggered for its trigger mode. This means that for our experiments, EMPIOT and the two DMMs start and end energy measurement at the same time.

3.3.1 Workloads

We tested this on devices which perform three main operations (i) sleep, (ii) software encryption, and (iii) packet transmission (listening and sending). With these different operations, we introduce three types of loads.

- Workload 1: All 3 operations,
- Workload 2: Send and sleep operations,
- Workload 3: Encryption and sleep operations,
- Workload 4: Encryption and send operations.

3.3.2 IoT Devices

We use five different IoT boards each with different energy characteristics. We used the TI SensorTag CC2650 which includes an ARM Cortex-M3 processor and supports the IEEE 802.15.4 standard. The TI Sensortag comes with TI-RTOS which we used for application development. The energy consumption of the board in sleep mode is very low powered (10μA) so we used it as a benchmark for profiling the energy of very low-power devices. To generate fast and temporal variations in power we used four 802.11 based IoT devices. These were the Avnet BCM4343W, Cypress CYW43907, RPiZW, and RPi3. The Avnet board includes an ARM Cortex-M4 processor and supports 802.11a/g/n. In power save mode, the minimum energy consumption of the Avnet board is around 10mA, processing consumes about 40mA, and packet transmission results spike up to 350mA. The Cypress board includes an AM Cortex R4 pocessor and supports 802.11a/g/n. Since the board also includes other components like an Ethernet chip, its sleep power is about 96mA. The processing power is about 140mA, and packet transmissions increase power consumption up to 400mA. On the Avnet and Cypress devices we used Free-RTOS and Wiced Studio for software development. Finally, to generate higher variations in power, the developed software enables power save mode (PS-Poll) mechanism so that the radio transitions into sleep mode while it is not transmitting. At certain intervals the radio wakes up and ping packets are transmitted as fast as possible. The packets' small size results in short and fast spikes in power consumption during transmission.

For RPiZW, the base and processing power are about 130mA and 180mA, respectively, and 802.11 transmissions result in spikes as high as 300mA. For the RPi3, the sleep and processing currents are about 280mA and 330mA, respectively, and 802.11 transmissions increase current consumption up to 500mA. A C program controls the transition between the three provided operations.

3.3.3 Results

Figure 3.10 shows the energy consumption of CC2650 and BCM4343W. CC2650 shows a clear transition between the three states while BCM4343W presents spikes across the trace. We can explain this observation as 802.11 communication requires communication with an access point. During power save mode in 802.11, the device wakes up every 100ms to receive the beacon packets generated by the access point.

Figure 3.9 presents power measurement error, computed as $[E - E1]/E * 100$ (for E = energy) when using the EMPIOT shield and the breakout board. Each marker is the median of 10 experiments where an experiment is 30 seconds long. The results we achieved indicate that EMPIOT is an accurate power measurement platform. Comparing subfigures (a) through (e), shoes that energy measurement error is higher when measuring very small currents. While the measurement error for 802.11-based boards is less than 2.5%, the error is less than 3.5% for the SensorTag. Considering Workload 1 on SensorTag, we observed that more than 40% of variations in current are less than 1mA. However, since the current resolution of INA219 is 100uA, these variations are not captured with a very fine granularity. For example, EMPIOT cannot detect current variations between 3.25mA and 3.27mA because these variations are less than 100uA. Therefore, as the power consumption of the SensorTag is significantly lower than that of other boards, small errors in current measurement result in a more considerable effect on total measurement error.

Figure 3.2 also shows the effect of calibration on accuracy. More specifically, since the measurement error of the breakout board is higher than that of EMPIOT's shield, calibrating the platform has shown to improve accuracy.

Figure 3.9 also shows that the measurement error of 9-bit resolution is slightly higher than that of 12-bit resolution. This is also particularly obvious for the SensorTag measurements. Compared with the 802.11 devices, SensorTag generates smaller variations in power which means that a higher resolution is required to capture the changes. Compared with 12-bit, using 9-bit resolution enhances the sampling rate.

3.3.4 Importance of Voltage Measurement

Most existing energy measurement platforms ignore the effect of voltage variations on energy measurement and instead use a fixed voltage instead to calculate energy. For this reason one of the aspects we wanted to test was whether voltage measurement had a significant effect on accuracy. This is also important because reading

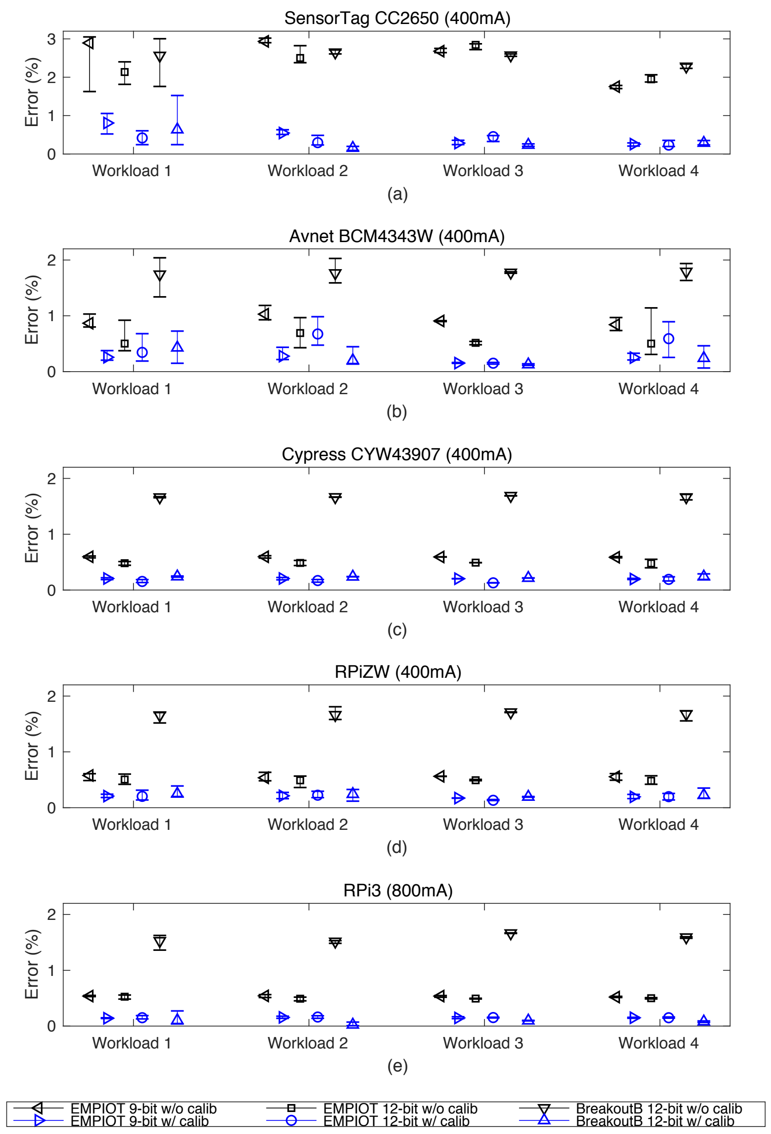


Figure 3.9: The energy measurement error of EMPIOT. Error bars show median, lower quartile and higher quartile. Each marker is the median of ten experiments where each is 30 seconds long.

from the voltage register takes extra time. If we can attain more accurate measurements with a fixed voltage, we can increase sampling rate. To verify this, we used Workload 1 to compute energy for two cases - (i) the average value of voltage measurements is used, (ii) the voltage samples are used. Error is computed as the difference between these cases and Figure 3.11 shows the results when we use a

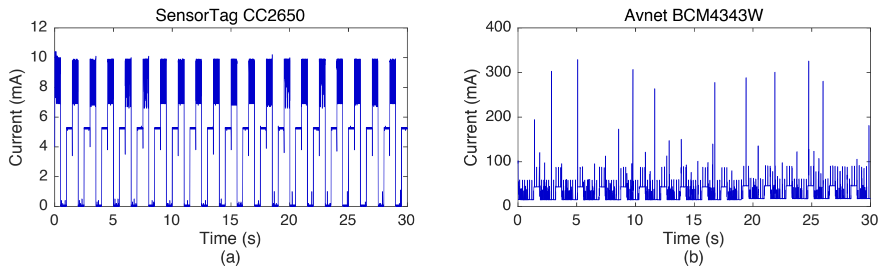


Figure 3.10: The energy trace of SensorTag CC2650 and Avnet BCM4343W when transitioning between sleep, encryption and transmission.

power supply and a battery for the devices we tested. These results indicate that neglecting voltage results in up to 0.45% increase in energy measurement error. The impact of voltage on energy measurement also depends on various factors including the stability of power source, the number and intensity of sudden increases in current draw, and the electronic characteristics of the device such as voltage stabilization. For example, when a battery is used, sudden variations of current result in higher measurement error when voltage is ignored. These results also show that both Cypress and Avnet devices cause significant variations in input voltage, compared with RPi3 and RPiZW. Our studies show that these variations are caused by the operations of 802.11 RF transceiver, thereby highlighting the importance of including voltage for 802.11-based IoT devices. On the other hand, the energy consumption of SensorTag and its 802.15.4 radio does not cause any significant variation in voltage.

3.3.5 Limitations

There are some limitations with the EMPIOT platform. One limitation is that its current measurement resolution is 100μA. This is especially problematic when measuring the SensorTag's energy consumption as its sleep current is 10uA. As a result, in order to improve the accuracy of energy measurement, we simply record

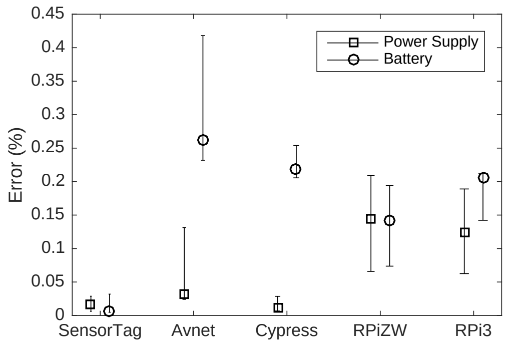


Figure 3.11: The errors resulting from ignoring voltage in calculating the energy measurement. Error bars show mean, upper quartile, and lower quartile.

10μA whenever the reported current value is around 100μA. However, this mechanism does not work for IoT devices that support multiple low-power modes. For example, EMPIOT is unable to capture transitions between 1μA, 10μ, and 100μA low-power states. Since the number of low-power states are usually limited, we can overcome this issue by explicitly informing EMPIOT about the transition through different low-overhead mechanisms such as pin toggling or serial communication. Doing this results in accounting for these missed transitions which can potentially increase accuracy.

Another limitation of EMPIOT is its warm-up time. Out studies show that the first 3 to 5 samples collected after initialization are not reliable. This means that we cannot measure for the first 5ms of an operation. However, we can address this issue in the software by ignoring the first few samples collected when energy measurement is inactive instead of turning on the shield board per measurement.

Algorithm 1: EMPIOT's software which is responsible for communicating with the shield, aggregating values, and storing them to file.

```
function main()
    setup the $I^2C$ driver (BCM/Linux) and bus speed;
    configure the INA219 gain and resolution;
    create thread sampler;
    create thread sample_writer;
    return;
thread sampler()
    while sampling is enabled do
        while conversion ready == 0 do
            check the bit;
        read bus voltage and shunt voltage values;
        compute_energy(prev_smp, new_smp);
        if use two buffers then
            if use_buffer1 == true then
                add entry to buffer1;
                if buffer1 is full then
                    signal the sample_writer thread;
                    use_buffer1 == false;
            else
                add entry to the buffer2;
                if buffer2 is full then
                    signal the sample_writer thread;
                    use_buffer1 == true;
        else if use a circular buffer then
            lock the buffer;
            add entry to the buffer;
            unlock the buffer;
            signal the sample_writer thread;
thread sample_writer()
    if use two buffers then
        if use_buffer1 == true then
            flush buffer1 to the file;
        else
            flush buffer2 to the file;
    else if use a circular buffer then
        lock the buffer;
        write entry to the file;
        unlock the buffer;
function compute_energy(prev_smp, new_smp)
    prv_power = (prv_smp.voltage) × (prv_smp.current);
    new_power = (new_smp.voltage) × (new_smp.current);
    time_diff = (new_smp.time) × (prv_smp.time);
    new_energy = new_power × time_diff ;
    tri = (new_power − prv_power) × time_diff / 2 ;
    new_energy -= tri;
    energy += new_energy;
```

Table 3.1: Conversion ready bit polling rate using BCM and Linux drivers for 12 and 9-bit resolution.

		Polling Rate				Conversion Time
		2500KHz	800KHz	500KHz	200KHz	
12-bit	BCM	45	15	9	3	1019\15
	Linux	23	9	6	2	
9-bit	BCM	9	2	1	1	245\9
	Linux	4	1	1	1	

Table 3.2: Calibration Values

EMPIOT	400mA/3.3V	$i_e = f_A^{-1}(i_a) = 0.9957\ i_a$
	400mA/5V	$i_e = f_A^{-1}(i_a) = 0.9963\ i_a$
	800mA/3.3V	$i_e = f_A^{-1}(i_a) = 0.9949\ i_a$
	800mA/5V	$i_e = f_A^{-1}(i_a) = 0.9956\ i_a$
Breakout Board	400mA/3.3V	$i_e = f_A^{-1}(i_a) = 0.9853\ i_a$
	400mA/5V	$i_e = f_A^{-1}(i_a) = 0.9869\ i_a$
	800mA/3.3V	$i_e = f_A^{-1}(i_a) = 0.0079\ i_a^2 + 0.9816 i_a$
	800mA/5V	$i_e = f_A^{-1}(i_a) = 0.0074\ i_a^2 + 0.982 i_a$

Chapter 4

Optimizing Boot Time in Linux Based IoT Devices

In this chapter we analyze the Linux boot up and shutdown process on an RPi and propose a novel framework towards improving it for duty cycled applications. In Section 4.1 we provide an in-depth description of the Linux boot up process. We present the components of our testbed and experimentation methodology in Section 4.2. In Section 4.3 we profile the Linux boot up process and measure the effect of unit configuration and flash memory on the duration and energy consumption of boot up. The Linux shutdown phase is studied in Section 4.4. The operation and performance evaluation of Pallex is presented Section 4.5.

4.1 Boot Up Process

As demonstrated in Figure 4.1, the Linux boot up process consists of three main phases: (i) the bootloader phase (P_{btl}), (ii) the kernel phase (P_{knl}), and (iii) the userspace initialization phase (P_{usi}). Each of these phases present unique opportunities to optimize boot time. In this paper we focus on P_{usi} because it enables the user to simply and effectively implement duty cycling to conserve energy.

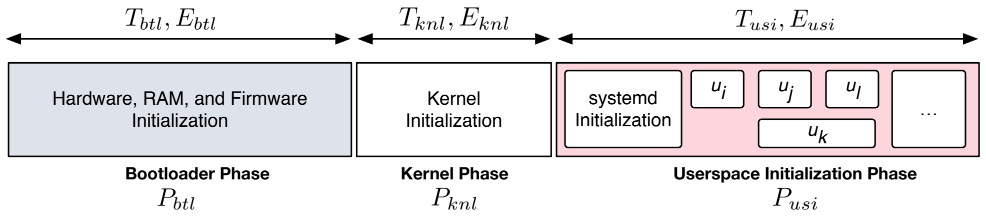


Figure 4.1: The sequence of operations during the boot up process. u_i, u_j, u_k and u_l refer to userspace units.

4.1.1 Bootloader Phase

In the Pi's boot up process, a two-stage bootloader[1] prepares the hardware to load the kernel. First, while the ARM processor is off, the GPU is powered up and initializes itself by executing a first stage bootloader that is burned into the SoC's ROM [69]. This stage instructs the GPU to power on the Secure Digital Card (SDC) and read a file called `bootcode.bin` from the first partition of the SDC. The execution of `bootcode.bin` enables the on-board SDRAM and loads `start.elf`, which contains firmware for the GPU. After reading in system configuration parameters from `config.txt`, the GPU loads the kernel image (`kernel.img`) along with kernel parameters (`cmdline.txt`) into the shared RAM allocated to the ARM processor. Lastly, the second stage powers on the ARM processor by triggering the reset signal [63]. Now the system is running the Linux kernel. In this paper we refer to the duration and energy consumption of this phase as T_{btl} and E_{btl}, respectively. Also, we refer to the ARM processor as ”processor”.

4.1.2 Kernel Phase

The Linux kernel handles all OS-related processes such as memory management, process scheduling, driver initialization, and overall system control. The kernel

[1]Until October 2012, the RPi platform used a three stage bootloader, with an additional file, `loader.bin`, executed by the GPU between the `bootcode.bin` and `start.elf` stages.

is initialized in two steps: The first step occurs when the kernel is loaded into memory and decompressed. Basic memory management is enabled during this stage as well. The next major step the kernel takes is launching an `init` process to run and transition to P_{usi}. In this paper we refer to the duration and energy consumption of the kernel phase as T_{knl} and E_{knl}, respectively.

4.1.3 Userspace Initialization Phase

The final phase of the boot up process is userspace initialization. In this paper we refer to the duration and energy consumption of this phase as T_{usi} and E_{usi}, respectively. During this phase, the units that are activated (a.k.a., initialized) in the userspace as well as the daemons that run in the background during active mode are activated by an `init` process. Most modern Linux distributions (including RSL) use `systemd` [70, 71, 72] as their initialization system. The predecessor to `systemd` was `System V init` (`sysvinit`), which traces its origins back to the original commercial Unix system. Compared to `sysvinit`, `systemd` offers advantages such as calendar-based job timers, a more unified API, and backward compatibility with `sysvinit`.

`systemd` is the first daemon to start during boot up and the last to exit during shutdown. In addition to operating processes and services, `systemd` is capable of triggering filesystem mounts, monitoring network sockets and running timers. Each of these capabilities is described by a set of configurations files, termed `unit` files. Unit types include: service units, which manage background services; mount units, to mount filesystems; and target units: to group and control other units. There are other unit types as well, but the details of their implementation reside outside the scope of this paper. In order to manage the dependencies and ordering of unit activation, unit files employ the syntax provided by `systemd`. When `sys-temd` is initialized, it first loads the unit configurations and determines the boot up goal. Based on the specified hierarchy of dependencies, `systemd` activates the units in order to reach the target goal. On RSL, the `rc.local` file is loaded and executed by `systemd` after all the services have been initialized. Therefore, the Raspberry Pi Foundation recommends general consumers launch user processes through this file to ensure all necessary hardware and software components are initialized. However, performance improvements are easily achieved through manually resolving dependencies and writing custom unit files, as we will demonstrate in Section 4.3.

One important feature `systemd` provides is the ability to activate units in parallel. This feature can save time compared to initializing units sequentially, even on a single-core or single-threaded board, as some units require time for hardware initialization [73]. Finally, `systemd` enables developers to customize rules for automatically starting, reloading, and killing services.

4.2 Testbed Overview

Figure 4.2 shows the architecture of the testbed developed to conduct experiments. It consists primarily of two hardware components: a *master* and a *minion*. The *master* is composed of a RPi3 and uses a shield board [74] for energy measurement. The *minion* refers to the device under test. We used two different minion boards in this paper, a RPi3 and a RPiZW, where both run the March 2018 release of RSL.

The *master* runs two programs: (i) an energy measurement program, and (ii) a control program that is responsible for enabling and disabling input power to

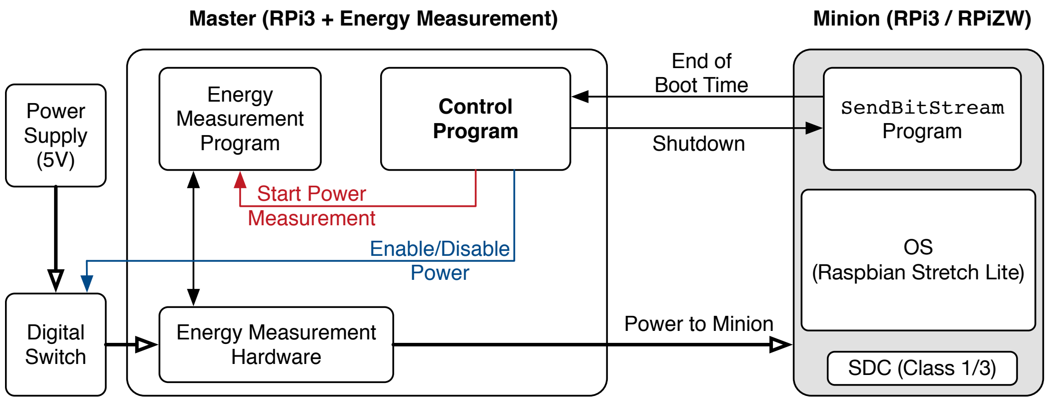


Figure 4.2: The schematic of the testbed used for measuring the duration and energy consumption of the boot up phases.

the minion. Algorithm 2 shows the pseudo-code of the control program running on the master device. This program is used to control power supply to the minion and measure the duration and energy consumption of the boot up process. First, the BCM [75] and WiringPi [76] libraries are initialized. Then, the energy measurement program is initialized to listen on a socket and receive commands from the control software. At the beginning of each experiment, the control program communicates with the energy measurement program through the socket to start power measurement. At the same time, the control program communicates with the digital switch through a GPIO pin controlled by the BCM library to turn on the minion. After the start of the boot up process, the control program uses the WiringPi library to detect and decode a message received from the minion when it finishes its operation. The end of the operation depends on the experimentation scenario and refers to cases such as the end of P_{usi} or the completion of a user application. The control program logs the duration and energy consumption of this operation for each experiment. Figure 4.3 shows the states of the minion during one single experiment from the instance the minion is powered on until the instance the minion sends the signal to the master after it has finished its operation.

It should be noted that the minion cannot use Ethernet, WiFi, or Bluetooth to inform the master about the end of its operation because their relevant units might not be active in some scenarios, as we will see later in this paper. Furthermore, it is not possible to use any of these mechanisms when the performance of P_{btl} and P_{knl} is being measured. To address this challenge, we send a message over a GPIO pin from the minion to the master. When the minion completes its operation, it runs a program that generates a simple bit pattern to notify the master. The minion generates a bit pattern, instead of a simple rising or falling edge signal, because of the GPIO voltage variations during the boot up process. Therefore, the generated bit pattern avoids the master from reporting false positives. This approach enables us to measure the performance of P_{btl} and P_{knl} because `systemd` is initialized immediately after the kernel phase, and GPIOs can be used as soon as `systemd` is initialized. The pattern generation program, named `SendBitStream`, is a unit that is activated by `systemd` once the targeted state (based on the experiment type) has been reached. Specifically, to measure $T_{btl}+T_{knl}$ (and $E_{btl}+E_{knl}$), this unit is called immediately after `systemd` initialization. This is achieved by creating a new unit without enforcing any dependencies. Similarly, to measure $T_{btl} + T_{knl} + T_{usi}$, this unit is called when all the required units have been activated. In order to measure boot up duration and energy until a particular unit has been activated, this unit is activated when its dependencies have been resolved. We used `systemd-analyze` to extract T_{knl}. By using this value, we can compute T_{btl} as well. A similar approach is used to measure the energy consumption of these phases.

For measuring energy consumption, we used EMPIOT, which measures voltage variations in the range of 0 to 5V and measures current variations up to 1A.

Algorithm 2: Master's control program

```
function startExperiments()
    /*to control power switch through a GPIO pin and receive bit pattern
      from the minion on another GPIO pin */
    setup the BCM and WiringPi GPIO libraries;
    /*setup energy measurement program and make it ready to measure
      power */
    initialize the energy measurement program;
    for $i = 0; i < 50; i++$ do
        cut power to the minion;
        apply power to the minion and start time/energy measurement;
        wait for minion to transmit bit pattern;
        record duration and energy consumption;
```

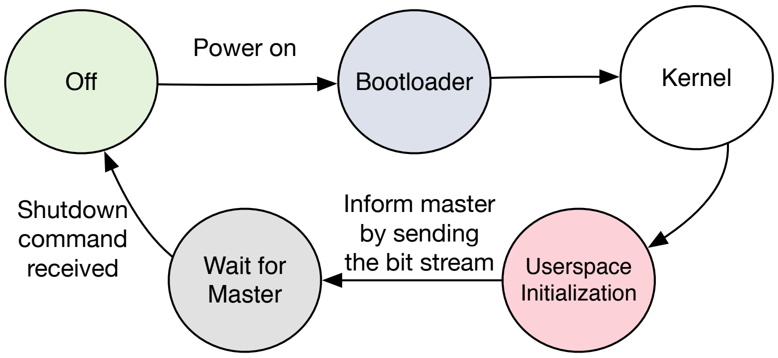


Figure 4.3: The state machine of the minion device during an experiment. Please note that during the wait state the system might be activating more units depending on the activation time of `SendBitStream`.

EMPIOT is capable of supersampling approximately 500,000 readings per second to data points streamed at 1KHz. The current and voltage resolution of this platform are 100μA and 4mV, respectively, when the 12-bit resolution mode is configured. The flexibility of this platform allows us to integrate it with our testbed.

Figure 4.4 shows the actual testbed used. In addition to the master node and the two minion boards, this testbed includes a gateway. Since for most of the scenarios, both wired and wireless communication interfaces are disabled, we need to use the serial port to communicate with the minions and configure the tests.

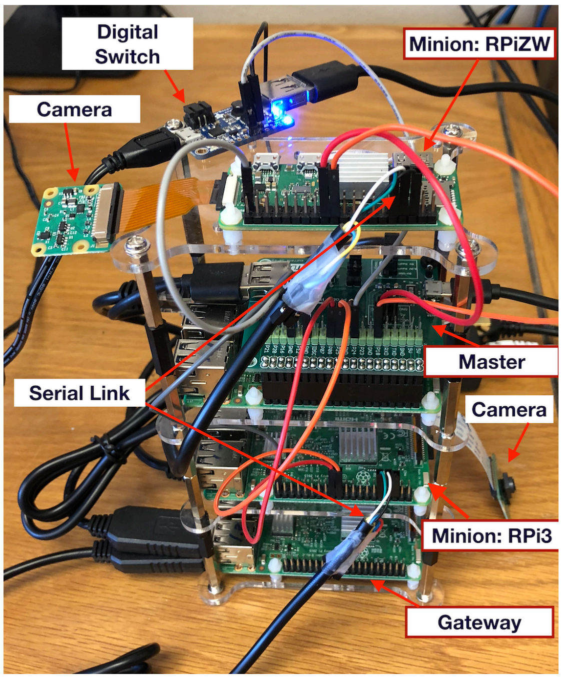


Figure 4.4: The hardware components of the testbed.

To this end, the gateway, which is always connected to our wired network, enables us to use UART and communicate with the minions. For the rest of this paper we refer to a minion board simply as "board", which is the device under test. Please note that in all the figures, each marker represents the mean and each error bar represents 95% confidence interval.

4.3 Profiling and Enhancement of the Linux Boot Up Process

In this section we first study the activation time of units during P_{usi}. We then profile system resource utilization in terms of processing, memory and I/O. In addition, we evaluate the effect of SDC speed and capacity usage on the duration and energy consumption of the three phases of the boot up process. Finally, we show how customizing userspace units can be employed to improve boot up duration and energy consumption.

4.3.1 A Deeper Look into the Userspace Initialization Phase

The userspace initialization phase activates a variety of units supporting different functionalities of the system. A summary of units is available in Appendix A.1. In the beginning of this phase, `systemd` is initialized and it loads unit configuration files to determine which units must be activated. It then creates a dependency tree to determine the ordering of unit dependency resolution. Units can be initialized in parallel, with respect to their dependency relationships, in order to improve efficiency.

Figure 4.5(a) and (b) show the activation duration of units during P_{usi} for the RPi3 and RPiZW, respectively. We have used the `systemd-analyze blame` utility to extract these data. Although all of the units are enabled for these experiments, we disabled WiFi and Ethernet connectivity in order to extract the activation duration of units without them being affected by external factors such as communicating with a WiFi access point. In addition, to focus on units that significantly contribute to T_{usi}, these figures do not display units that require less than 10ms to complete activation. Please note that the x-axis of both Figure 4.5(a) and (b) start at $t = 6s$ because $T_{btl} = 3.65s$ and $T_{knl} = 2.85s$ for both boards.

For the RPi3, our results show that the system units requiring the highest overhead are `dev-mmcblk0p2.device`, `networking.service`, `hciuart.service`, and `systemd-resolved.service`. Except the first unit, which is responsible for

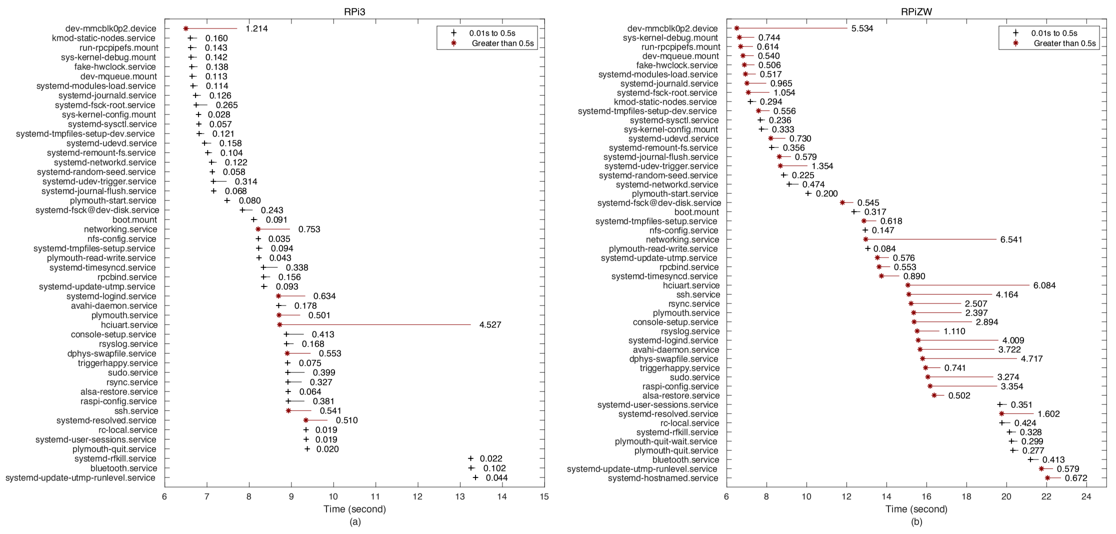


Figure 4.5: The starting time and initialization duration of units during P_{usi} for (a) RPi3 and (b) RPiZW. Only the units with activation duration longer than 10ms are included in this figure.

bringing the root partition into the scope of `systemd`, the rest are networking-related services. Networking services generally require a longer activation duration compared to other units because they require a combination of initializing hardware and networking utilities. The `hciuart.service` is responsible for the initialization of Host Controller Interface (HCI) to provide a uniform interface for accessing Bluetooth hardware capabilities. All USB-Bluetooth adapters operate with a HCI interface over the USB link. During its initialization, HCI creates read and write communication threads, establishes a connection to the Bluetooth transceiver, and reads device buffer sizes. Enabling real-time Bluetooth communication requires the processor to frequently context-switch to monitor UART communications. Because `bluetooth.service` depends on `hciuart.service`, it finishes initialization after `hciuart.service`. An interesting behavior of these services is that they both complete their activation after `rc.local`. This means that user applications that rely on Bluetooth cannot be started using `rc.local`. Instead, these applications require a `systemd` unit that is scheduled to be activated after the completion of `bluetooth.service`.

On the RPiZW, we observe that T_{usi} is about 17s, which is 10s longer than that of the RPi3. The same units mentioned for the RPi3 consume most of the userspace initialization time on the RPiZW as well. Since the RPiZW's processor only has one core, compared to the RPi3's quad-core processor, the difference in duration is expected. A multi-core processor can parallelize unit activations across multiple cores, while a single-core processor must context-switch more frequently between tasks. This also explains why different sets of units are presented in Figure 4.5(a) and (b). Furthermore, other units including `systemd-login.service`, `console-setup.service`, `alsa-restore.service`, and `systemd-user-sessions.service` reportedly require a longer activation duration (at least 1s) on the RPiZW.

Our analysis of unit activation duration revealed one significant shortcoming about `systemd-analyze blame`. The initialization duration calculated by this utility is not completely accurate if the dependency tree is not precisely configured. For example, if `systemd` attempts to activate a fast unit that depends on a longer unit which has not been activated yet, the faster unit cannot complete its activation until the dependency has been resolved. More specifically, if the activation duration of longer and shorter units are 800ms and 10ms, respectively, then `systemd-analyze blame` reports 810ms as the initialization duration of the faster unit. For example, for `sudo.service`, `systemd-analyze blame` falsely reports a long initialization duration because `systemd` attempts to activate it before the filesystem the service depends on is mounted. Therefore, the values generated by this utility may be longer than the actual activation duration of each unit, but these values are not shorter. To cover all the units that contribute significantly to the boot up process, we have included only those units that require more than 10ms according to `systemd-analyze blame`. In Figure 4.5, we include Type I errors to avoid excluding any units that require more than 10ms. Later, to eliminate Type I errors, we identify and study the impact of units that significantly affect the boot up process.

4.3.2 Customizing Userspace Initialization

In this section, we analyze the effect of disabling optional units on the performance of the userspace initialization phase. This is referred to as *unit configuration* in this paper. The `systemctl` [77] utility is used to implement unit configuration.

Among the units activated during P_{usi}, some of them are essential to maintain stable system operation. For example, units that are responsible for mounting the file systems and loading kernel modules cannot be safely disabled. Another example, nearly all user applications will work even if `systemd-random-seed.service` is disabled. However, disabling this service is a security risk, because it is critical for maintaining higher entropy for the secure generation of random numbers used in encryption algorithms. A complete list of these units, which are referred to as *Essential Units* (EU) in this paper, can be found in Appendix A.1. The next category includes a significant number of services and is referred to as *Networking-Related Services* (NRS) in this paper. Appendix A.1 overviews these services. These services are by far the most variable in terms of activation duration because they often rely on external dependencies (e.g., association with an access point, communicating with a DHCP server, etc.) and/or initializing physical hardware such as the WiFi and Bluetooth transceivers. Due to the significant effect of NRS on boot up performance, we study the effect of following unit configurations on P_{usi}:

– *EU*. Refers to the case where only the essential units (cf. Appendix A.1.1) are enabled. For EU configuration, we detect the end of userspace initialization when `rc.local` runs because it is the last unit file that is executed.

- *MMS*. Refers to `dphys-swapfile.service`. For configuration EU w/ MMS, we detect the end of userspace initialization when `rc.local` runs because it is the last unit file that is executed.

- *NET1*. Refers to `networking.service`. For configuration EU w/ NET1, we detect the end of userspace initialization when `rc.local` runs because it is the last unit file that is executed.

- *NET2*. Refers to `networking.service` and `sshd.service`. For configuration EU w/ NET2, we detect the end of userspace initialization when `rc.local` runs because it is the last unit file that is executed.

- *NET3*. Refers to `bluetoothd.service` and `hciuart.service`. For configuration EU w/ NET3, since `bluetooth.service` is the last service that is executed, we detect the end of userspace initialization when this service completes its initialization.

- *ALLU*. Refers to the case where all units are enabled. For this configuration, since `bluetooth.service` is the last service that is executed, we detect the end of userspace initialization when this service is activated. For configuration ALLU w/o NET3, we detect the end of userspace initialization when `rc.local` runs because it is the last unit file that is executed.

In addition to the classifications detailed above, since the actual association of an RPi with an access point introduces more variations due to the control messages exchanged between the two parties, we report the results separately for the cases where a WiFi connection is established. Figure 4.6(a) and (b) show the

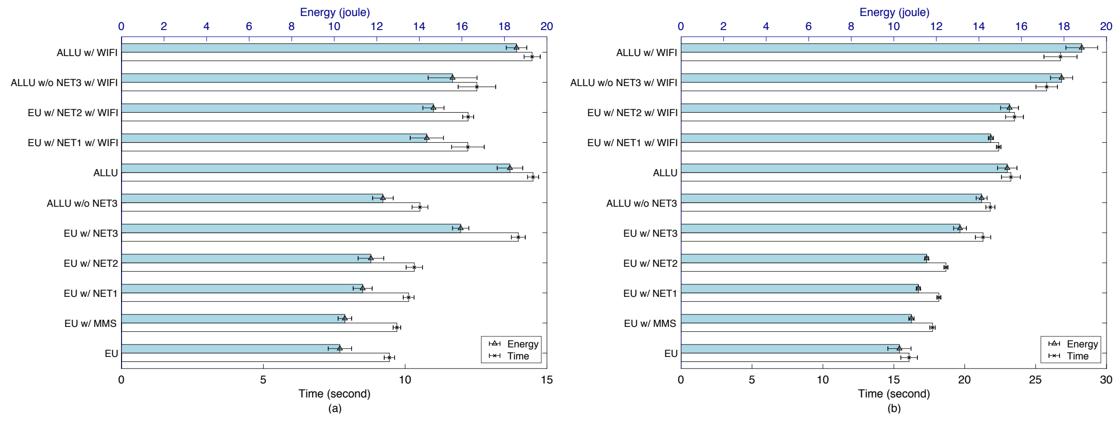


Figure 4.6: The duration and energy consumption of booting up the (a) RPi3 and (b) RPiZW for different unit configurations. The lower and upper axes show $T_{btl} + T_{knl} + T_{usi}$ and $E_{btl} + E_{knl} + E_{usi}$, respectively.

impact of unit configuration on both the RPi3 and RPiZW, respectively, in terms of T_{usi} and E_{usi}.

These figures clearly show the benefits of unit configuration to enhance the performance of boot up phase. For example, applying unit configuration EU reduces the energy consumption by 43.62% compared to ALLU, for the RPi3 board. These results also reveal that WiFi significantly affects T_{usi} and E_{usi}. Specifically, for configuration ES w/NET1, enabling WiFi increases T_{usi} by around 5s and 13s, for the RPi3 and RPiZW, respectively. It must be noted that the exact increase depends on factors such as interference, channel congestion, and the load of the access point during the association process. For example, the duration of WPA authentication and IP allocation increases as the current load of the access point is intensified. Since these external dependencies are outside the scope of the RPi performance, unnecessary services must be disabled or careful attention must be paid to link quality and access point load to achieve a desirable performance.

These results also reveal the higher effect of `hciuart.service` and `bluetooth.service` on the RPi3. Since activating these services do not significantly utilize the processing resources of the RPi3 (as we will show in Section 4.3.3), the waste of processing resources, and hence energy consumption, is higher that than of RPiZW. Therefore, EU w/ NET3 compared to EU results in a 54.8% energy increase on RPi3, compared to the 26.92% increase on RPiZW. In addition, because of this, we can observe that on RPi3, which can initialize WiFi and NET3 services concurrently, the duration of ”ALLU” and ”ALLU w/ WiFi” configurations are almost equal. In contrast, for the RPiZW, the duration of ”ALLU w/ WiFi” is longer than ”ALLU” because the single-core processor needs to interleave the tasks of initializing WiFi and NET3 services. We will further study this behaviour in the next section.

It must be noted that *disabling* units does not necessarily prevent their activation by `systemd`. More specifically, units might be activated when: (i) other units relying on them are activated, or (ii) in the case of external event hooks such as attaching a device. For example, `alsa-restore.service` is automatically activated even if it has been disabled. However, as its main function is to initialize the onboard sound card, it is not required by most IoT applications. Therefore, it is worth *masking* this service using `systemctl`. This is performed by pointing the unit file to the special device `/dev/null` so that the dependency tree can mark it as resolved for dependants without actually running it.

4.3.3 Resource Utilization During Userspace Initialization

To justify running a user application in parallel during P_{usi}, we study resource utilization during this phase. To this end, we measured processor utilization, memory utilization, and SDC I/O speed on the RPi3 and RPiZW. Resource monitoring is performed by a shell program that starts as soon as `systemd` is initialized. For

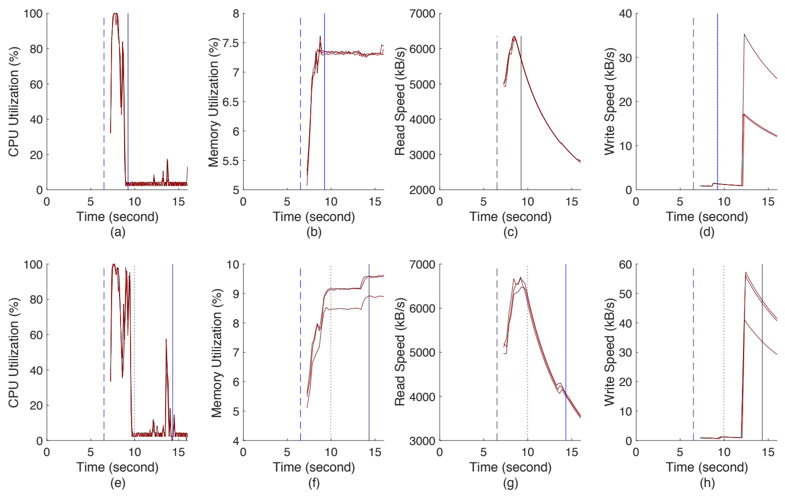


Figure 4.7: The resource utilization of the RPi3 during userspace initialization for three different experiments. Sub-figures (a)-(d) represent unit configuration EU, and sub-figures (e)-(h) represent unit configuration ALLU. Dashed lines indicate the start of P_{usi}, dotted lines indicate the instance `rc.local` is activated, and solid lines indicate the end of P_{usi}.

this reason we noticed that resource monitoring is not available for 0.75s and 1.2s after the completion of P_{knl} on the RPi3 and RPiZW, respectively. To record processor utilization, we wrote a `gawk` [78] script to read values from `/proc/stat` and calculate the current percentage of processor utilization across all cores with high granularity and low overhead. We read directly from `/proc/meminfo` to determine memory utilization, and used the `iostat` [79] utility for collecting SDC I/O utilization. Figures 4.7 and 4.8 show the results for three trials.

Comparing the two figures indicates the significantly higher processor utilization of the RPiZW compared to the RPi3. For the RPi3, we notice that processor utilization drops to less than 5% as soon as `rc.local` is invoked, regardless of the unit configuration applied (compare Figure 4.7(a) and (e)). For the RPiZW, however, when unit configuration ALLU is applied, processor utilization drops to around 5% after the end of userspace initialization, which is the completion of NET3 services (compare Figure 4.8(a) and (e)). We justify this behavior by

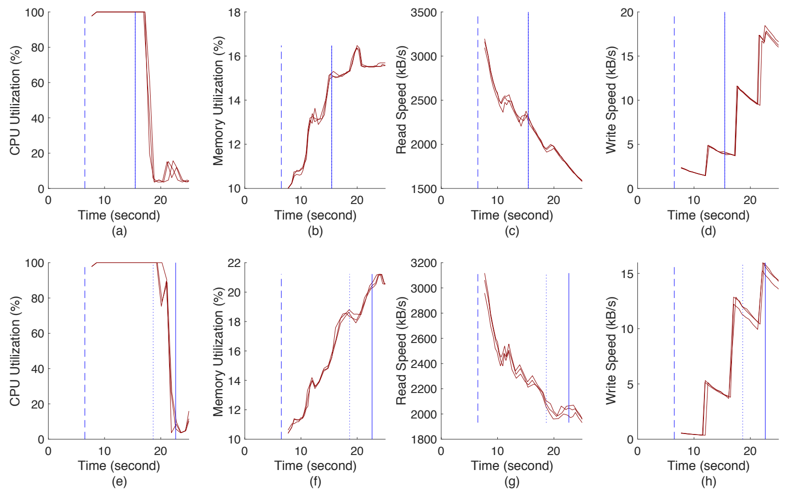


Figure 4.8: The resource utilization of the RPiZW during userspace initialization for 3 different experiments. Sub-figures (a)-(d) represent unit configuration EU, and sub-figures (e)-(h) represent unit configuration ALLU. Dashed lines indicate the start of P_{usi}, dotted lines indicate the instance `rc.local` is activated, and solid lines indicate the end of P_{usi}.

referring back to Figure 4.5. While the RPi3 is almost finished with userspace initialization (except the NET3 services) at the time `rc.local` is activated, RPiZW needs to complete the activation of multiple units. Specifically, as Figure 4.5 shows, a considerable number of units with loading time longer than 10ms are being activated around $t = 20$.

For RPi3, enabling all services increases memory utilization from around 7% to 10%. For RPiZW, the increase is from around 16% to 21%. These results indicate that, even for the unit configuration ALLU, more than 900MB and 400MB of RAM is available on the RPi3 and RPiZW, respectively. In terms of I/O, since the RPi3 initializes more units in parallel, its I/O speed is almost double that of the RPiZW. Although the processor utilization of the RPiZW is around 100% throughout P_{usi}, we can still benefit from concurrent execution of user applications within this phase if they mostly rely on peripheral initialization and I/O operations. A sample IoT application that satisfies this requirement is capturing a photo using a camera. For example, the RPi camera module [80] connects over a Camera Serial Interface (CSI) to the GPU on the RPi. While a picture is being captured and processed, the processor can switch to other tasks, as there are several steps performed by the camera that are independent of the RPi's processor. In particular, the camera has the physical requirement of exposure time. Next, the camera needs to process the image. According to the manufacturer of the camera module's image sensor (IMX219PQ [81]), the camera has Lens Shading Correction (LSC) functionality, which means the image undergoes some processing on the camera module before it is sent over CSI. After this step, the image must be sent over the CSI interface as a series of Bayer frames to the RPi's GPU. Next, the GPU's VideoCore firmware assembles the image. Finally, the processor can receive the assembled image from the GPU. In Section 4.5 we will show the performance improvement of this application running during P_{usi}.

4.3.4 Profiling the Time and Energy Consumption of Bootloader and Kernel Phases

In this section we study the duration and energy consumption of bootloader, kernel, and userspace initialization phases versus the properties of the SDC used. In order to measure the effect of SDC speed on performance, we used two 32GB Sandisk SDCs: (i) a UHS (Ultra High Speed) class 1, and (ii) a UHS class 3. Note that UHS 1 and UHS 3 refer to minimum write speed 10 MB/sec and 30 MB/sec, respectively. In addition to SDC speed, we are interested in measuring the effect of SDC capacity utilization on boot up performance. Modern SDCs are implemented with NAND technology. Compared to NOR technology, NAND offers lower power consumption, lower cost per bit, higher density and faster write speed. However, as the disk fills up, write performance starts to degrade. In order to measure the effect of SDC utilization on boot up performance, we fill 5%, 50%, and 95% of the SDC capacity. To fill the SDC with data, we used the `dd` [82] utility which allows us to write blocks of memory to the SDC.

Tables II and III demonstrate the results averaged over 50 trials for each configuration. The following unit configurations are used for these measurements: (i) EU, and (ii) ALLU w/o NET3 w/o WiFi). We excluded NET3 due to the high variations caused by these services. In terms of SDC capacity usage, these results show no effect on boot up performance. However, the use of faster SDC results in a slight reduction in energy consumption. For example, for the ALLU w/o NET3 configuration on the RPi3, on average the faster SDC reduces energy by 2.5%, compared to the slow SDC.

According to these results, regardless of the SDC type and capacity usage, T_{btl} and T_{knl} are similar for the two boards. However, the energy consumption of these phases is higher on the RPi3 than on the RPiZW. Compared to P_{usi}, which is process-intensive, P_{btl} and P_{knl} do not benefit from the higher processing power of RPi3's SoC because their main operation is to load the kernel and initialize hardware components, tasks that are mostly synchronous and not easily threaded across cores. Therefore, since the RPi3 has a more complex and powerful SoC, more resources are wasted on this board during the first two phases. In contrast, E_{usi} of the RPi3 is actually lower than that of the RPiZW. During P_{usi}, the RPi3 runs a larger number of processes in parallel, and therefore requires less energy to reach the boot up target by reducing the total amount of time spent in this phase.

Table 4.1: The duration and energy consumption of the bootloader phase ($T_{btl} + T_{knl}$, $E_{btl} + E_{knl}$), kernel phase (T_{usi}, E_{usi}), and userspace initialization phase (T_{usi}, E_{usi}) for RPi3. The left and right values in each cell show duration (second) and energy consumption (joule).

		EU			**ALLU w/o NET3 w/o WiFi**		
		Bootloader+Kernel	Userspace	Total	Bootloader+Kernel	Userspace	Total
Slow SDC	5%	6.5, 4.51	2.94, 5.77	9.44, 10.28	6.5, 4.51	4.04, 7.72	10.54, 12.23
	50%	6.5, 4.51	3.23, 5.95	9.73, 10.46	6.5, 4.51	3.72, 7.32	10.22, 11.74
	95%	6.5, 4.51	3.23, 5.89	9.73, 10.4	6.5, 4.51	3.72, 7.28	10.22, 11.79
	AVG	6.5, 4.51	3.13, 5.87	9.63, 10.38	6.5, 4.51	3.83, 8.91	10.33, 11.92
Fast SDC	5%	6.5, 4.51	2.94, 5.6	9.44, 10.11	6.5, 4.51	3.74, 7.13	10.24, 11.64
	50%	6.5, 4.51	2.97, 5.68	9.46, 10.19	6.5, 4.51	3.66, 7.07	10.16, 11.58
	95%	6.5, 4.51	2.96, 5.67	9.46, 10.18	6.5, 4.51	3.71, 7.2	10.21, 11.71
	AVG	6.5, 4.51	2.95, 5.65	9.45, 10.16	6.5, 4.51	3.7, 7.133	10.2, 11.64

Table 4.2: The duration and energy consumption of the bootloader phase ($T_{btl} + T_{knl}$, $E_{btl} + E_{knl}$), kernel phase (T_{usi}, E_{usi}), and userspace initialization phase (T_{usi}, E_{usi}) for RPiZW. The left and right values in each cell show duration (second) and energy consumption (joule).

		EU			**ALLU w/o NET3 w/o WiFi**		
		Bootloader+Kernel	Userspace	Total	Bootloader+Kernel	Userspace	Total
Slow SDC	5%	6.5, 3.26	9.73, 7.02	16.23, 10.28	6.5, 3.26	15.48, 10.91	21.88, 14.17
	50%	6.5, 3.26	9.43, 6.34	15.93, 9.6	6.5, 3.26	15.4, 10.97	21.9, 14.23
	95%	6.5, 3.26	9.8, 6.57	16.3, 9.83	6.5, 3.26	15.47, 10.94	21.97, 14.2
	AVG	6.5, 3.26	9.65, 6.64	16.15, 9.9	6.5, 3.26	15.45, 10.94	21.92, 14.2
Fast SDC	5%	6.5, 3.26	9.47, 6.56	15.97, 9.82	6.5, 3.26	15.3, 10.86	21.8, 14.12
	50%	6.5, 3.26	9.41, 6.31	15.91, 9.57	6.5, 3.26	15.39, 10.92	21.89, 14.18
	95%	6.5, 3.26	9.47, 6.35	15.97, 9.61	6.5, 3.26	15.34, 10.92	21.84, 14.18
	AVG	6.5, 3.26	9.45, 6.41	15.95, 9.67	6.5, 3.26	15.34, 10.9	21.84, 14.16

4.4 Profiling and Enhancement of the Linux Shutdown Phase

Throughout the scope of this paper, special attention is paid to system boot up phases rather than the shutdown phase. This is because the time and energy required for boot up is higher than that required for shutdown, and the gains are therefore more significant. However, there are several ways through which shutdown time and energy consumption can be reduced as well. The naive approach is to cut the power to the RPi as soon as the user application is completed. Unfortunately, cutting the power improperly might result in corrupting data blocks on SDC. If these blocks also happen to coincide with the sectors necessary for boot up or important files in the `rootfs`, the device could be rendered unrecoverable. In order to address this problem, all of the SDC's partitions can be mounted as read-only to guarantee there would be no file operations when a power loss event occurs. However, a read-only solution is complicated and not feasible when the user application needs to store or analyze large quantities of dynamic data such as running a machine learning algorithm. Furthermore, since the filesystems are by default read-only, updating the device becomes a difficult and lengthy process, requiring virtual root filesystems mounted to RAM disks and multiple remount operations on the SDC. In this case, if large amounts of data must be stored for processing or before transmission, external storage is required. This solution, however, introduces extra power consumption and might cancel out any gains achieved by a faster shutdown. Additionally, if no external device is used, workarounds must be implemented for system logging and other system functionalities, which rely on a writeable filesystem. This analysis is outside the scope of this paper.

The next approach is to make only the boot partition write-protected. However, this solution does not necessarily prevent data corruption, because flash partitions are not truly separated as they would be on a traditional hard drive. SDCs use a Flash Transition Layer (FTL) to map virtual file blocks to their actual location in the storage [83]. Many SDCs are preloaded with a wear-leveling firmware which uses the FTL to re-arrange data blocks, often mixing across partition lines (transparently to the RPi) in order to enhance block device lifetime. In this case, data corruption can occur if the power is cut while read-only data is being migrated during the wear-leveling operation. Therefore, mixing read-only and writeable partitions does not guarantee protection against improper shutdowns. The only way to guarantee it is safe to cut the device power is to guarantee that no operations are being performed on the SDC.

In the rest of this section, we present and evaluate two suitable approaches, *graceful shutdown* and *forced shutdown*, to power off a duty-cycled IoT system. Furthermore, we assess the tradeoffs between system reliability and energy consumption. Please note that T_{sdn} and E_{sdn} refer to the duration and energy consumption of the shutdown phase.

4.4.1 Graceful Shutdown

During a graceful shutdown, `systemd` sends a shutdown signal to all of the running processes. After these processes exit and the network interfaces are brought down, the filesystems are unmounted and power to the device is safely cut. The amount of time required to unmount the filesystems is almost fixed and beyond the control of the user. However, the lower the number of running processes which must return an exit code before `shutdown.target` is reached, the faster `systemd` can finalize the shutdown; therefore, removing extraneous units expedites the shutdown phase.

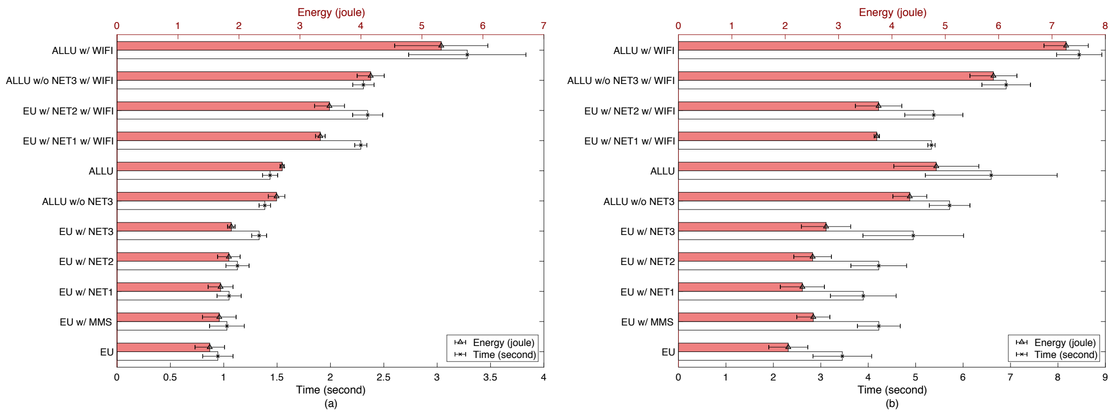


Figure 4.9: The duration and energy consumption of *graceful shutdown* for the (a) RPi3 and (b) RPiZW when various unit configurations are applied. The lower and upper axes show T_{sdn} and E_{sdn}, respectively.

This behavior is best exemplified in Figure 4.9. These figures represent the time and energy consumption of the RPi3 and RPiZW for various unit configurations.

When WiFi is disconnected, using unit configuration EU reduces energy consumption by 43.9% and 57.4%, on the RPi3 and RPiZW, respectively, compared to unit configuration ALLU. When WiFi is connected, using unit configuration EU reduces energy consumption by 37.3% and 48.85% for the RPi3 and RPiZW, respectively, compared to ALLU. These results also show the significant effect of WiFi communication on E_{sdn}. During the shutdown phase, the system invokes `ifdown` to ensure all connected networks are brought down properly and then powered off. This process takes around 0.8s when `avahi-daemon.service` is disabled. When this service is enabled, the duration varies depending on the network speed and configuration. In our testbed, we noticed a delay of up to 12s. For example, unit configuration EU w/ NET1 w/ WiFi increases energy consumption by 96.4% and 60%, for the RPi3 and RPiZW, respectively, compared to EU w/ NET1.

4.4.2 Forced Shutdown

Not all IoT applications require a clean shutdown to remain functional. There are two alternative commands, `systemctl halt -force` (equivalent to the traditional `halt` command) and `systemctl halt -force -force` (equivalent to the traditional `halt -f` command), which result in shorter shutdown phases. In this paper we refer to these approaches as *forced shutdown* and *forced-forced shutdown.* Both of these approaches skip the steps performed by `shutdown` to notify running processes of the impending shutdown and wait for them to gracefully exit. Therefore, steps such as recording the shutdown event and any `STDOUT` or `STDERR` output that would normally be printed to a log file during the shutdown phase are skipped by these commands. In the case of `systemctl halt -force`, this may be acceptable, as the logging of system shutdown events is not often critical to IoT applications, and any necessary shutdown logs may be generated manually by the user application. `systemctl halt -force` also properly disconnects from networking interfaces by calling `ifdown` on all connected interfaces.

Figure 4.10 shows the impact of using forced shutdown on both time and energy consumption. Comparing this figure against Figure 4.9 demonstrates the performance benefits of killing processes rather than gracefully terminating them. Although this mechanism results in lower energy consumption for both platforms, the effects are more apparent on the RPiZW. When gracefully shutting down, the RPi3 provides system processes with more resources to finalize their operations and terminate properly, resulting in a shorter shutdown phase. On the other hand, when forced shutdown is used, running processes are simply killed, which does not require significant system resources. However, the OS still waits for the networking interfaces to be brought down before continuing the shutdown phase. As a result, the extra power provided by the RPi3's processor is wasted, making the RPiZW

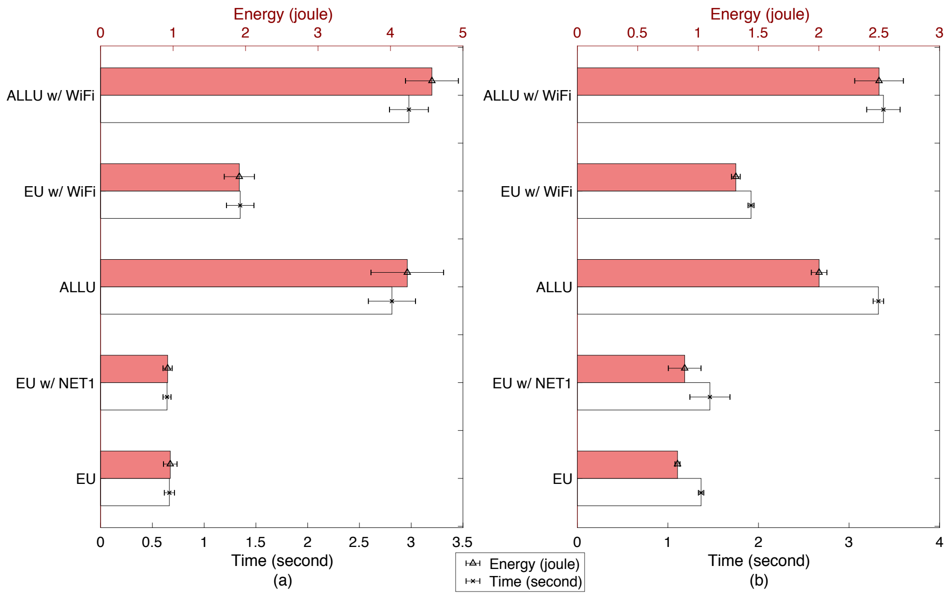


Figure 4.10: The duration and energy consumption of *forced shutdown* for the (a) RPi3 and (b) RPiZW when various unit configurations are applied. The lower and upper axes show T_{sdn} and E_{sdn}, respectively.

more energy efficient during the shutdown phase.

The second approach, `systemctl halt -force -force`, is faster than calling `systemctl halt -force` because the processes are not killed. Instead, the processes are simply abandoned as the processor cores are stopped. This command physically halts the processor and cuts power almost immediately without bringing down network interfaces or unmounting filesystems. Eliminating these steps reduces the shutdown duration of the RPi3 and RPiZW to less than 200ms and 400ms, respectively, and the energy consumption to less than 700mJ and 350mJ, respectively, as Figure 4.11 shows. Unfortunately, according to the documentation [77], using this command may cause data corruption. However, some approaches exist to minimize the risk. For example, all processes required or started by the user application can be killed manually first, and the `sync` command must be run to commit unsaved buffers to the SDC. For preventing issues related to wear-leveling, increasing the percentage of unused storage on SDC reduces the frequency

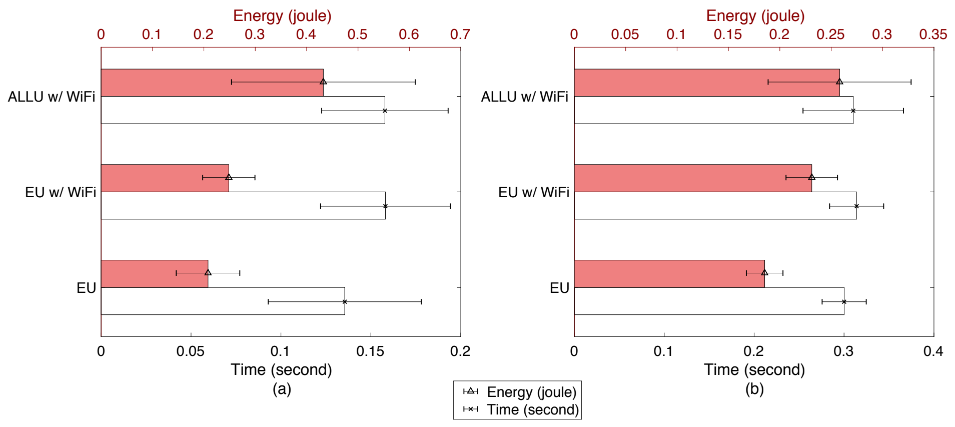


Figure 4.11: The duration and energy consumption of *forced-forced shutdown* for the (a) RPi3 and (b) RPiZW when various unit configurations are applied. The lower and upper axes show T_{sdn} and E_{sdn}, respectively.

of moving critical data blocks by the wear-leveling algorithm. However, since the chance of data corruption is not fully eliminated, it is up to the user to calculate the risk involved in shutting down the system using this mechanism repeatedly (or across many devices). These calculations are outside the scope of this paper. Additionally, for both forced-shutdown approaches, the user must consider running some processes (such as `fake-hwclock save` and `systemd-random-seed save`) manually to ensure system integrity and security.

It is worth mentioning that the `halt` system call for the ARM architecture automatically calls `machine_power_off()` to power off the board rather than entering the traditional *halted state* where the board stays powered on after the processor is powered off. Other architectures may require additional flags or even different commands in order to achieve a similar result.

4.5 Parallelizing Application Processes with Userspace Initialization

As we showed in the previous sections, the userspace initialization phase does not fully utilize the system resources. In this section, we propose *Pallex*, a parallel execution framework to run user applications during the userspace initialization phase. After presenting the implementation of Pallex and providing guidelines for applying this framework to various types of applications, we evaluate its performance considering various IoT application scenarios.

4.5.1 Pallex

The basic idea of Pallex is to divide a user application into stages and run each stage based on the set of available units and the completion of prerequisite stages. For a given user application, we break the code into a stage set S = $\{s_i, s_j, ...\}$. Each stage s_i has two types of dependencies:

$$\mathrm{D}(s_i) = \{s_j, s_k, ...\} \tag{4.1}$$

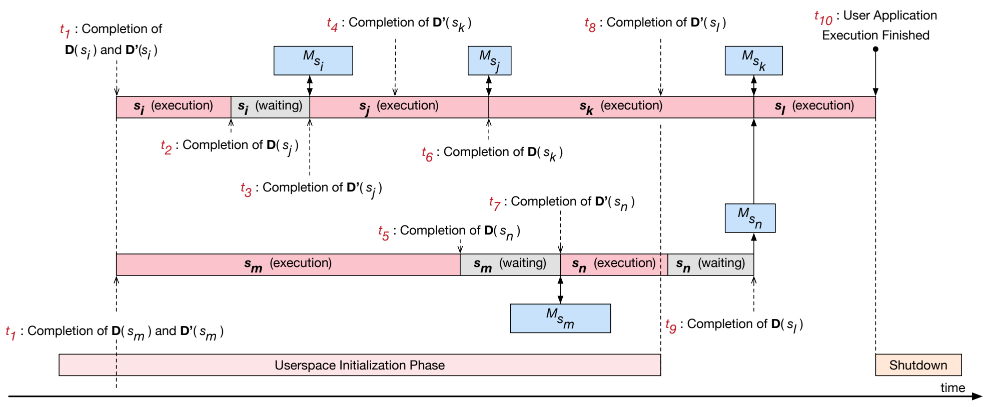
and

$$\mathrm{D}'(s_i) = \{u_j, u_k, ...\} \tag{4.2}$$

where $\mathrm{D}'(s_i)$ is called the *stage dependency set* and refers to the set of user application stages that must be completed before starting stage s_i, and $\mathrm{D}(s_i)$ is called *unit dependency set* and refers to the set of units on which stage s_i depends on. Each stage is a process invoked by `systemd` when the dependency sets specified in the stage's unit file are resolved. Therefore, for `systemd` to build the dependency tree and run the stages in an orderly manner, each stage's dependency sets must be specified in its corresponding unit file's `Requires` section.

Since the stages of execution are independent processes, they do not share the same address space. Therefore, a mechanism is required to share data across processes. To this end, Pallex utilizes the mechanism offered by *Unix Domain Socket* (UDS) [84]. UDS is a data communication endpoint for exchanging data between processes executing on the same host OS, and provides a standard method for implementing Inter-Process Communication (IPC) on a Unix-based system. Since UDS handles communication within the Linux kernel, it is initialized in the kernel space as well. This indicates that we can use them as reliable means of communication during P_{usi}. When a stage s_i completes its execution and needs to transfer data to another stage s_j, the kernel blocks s_i until s_j is ready [85] . During this time, processor resources are used for the activation of units that are required by stage s_j. When the dependencies of s_j are resolved and it requests the shared data, s_i completes the data transfer and then exits.

There are at least three other methods of IPCs we could have used, including TCP/UDP sockets, POSIX message queues, and writing to a file. The reasons that we did not use these methods are as follows. First, using a TCP/UDP connection over localhost requires waiting for the network services to load, which defeats the purpose of Pallex. Specifically, one of the largest delays in the userspace initialization is caused by the initialization of `networking.service`, as shown in Section 4.3. Second, writing to a file results in I/O overhead and affects the boot up time and message sharing delay because of the SDC activity during the userspace initialization phase. Lastly, we did not use POSIX message queues because they are too low-level and require careful configuration. Although both UDS and POSIX message queues are available almost concurrently, for the latter it is necessary to



t_1 : Completion of D(s_i) and D'(s_i)
t_4 : Completion of D'(s_k)
t_8 : Completion of D'(s_l)
t_{10} : User Application Execution Finished
M_{s_i}
M_{s_j}
M_{s_k}
s_i (execution)
s_i (waiting)
s_j (execution)
s_k (execution)
s_l (execution)
t_2 : Completion of D(s_j)
t_6 : Completion of D(s_k)
t_3 : Completion of D'(s_j)
t_7 : Completion of D'(s_n)
t_5 : Completion of D(s_n)
M_{s_n}
s_m (execution)
s_m (waiting)
s_n (execution)
s_n (waiting)
M_{s_m}
t_1 : Completion of D(s_m) and D'(s_m)
t_9 : Completion of D(s_l)
Userspace Initialization Phase
Shutdown
time

Figure 4.12: An example of Pallex. The *stage set* of user application includes six stages, $S = \{s_i, s_j, s_k, s_l, s_m, s_n\}$. A new stage is started as soon as its *stage dependency set* and *unit dependency set* are satisfied. Stages share their messages using the mechanism offered by Unix Domain Socket (UDS).

configure the size and number of messages and their accepted/waiting status to make sure that enough buffer is available. Although message queues offer various features, UDS is standard, easy to use, and fast. UDS allow us to send data to the socket before the receiving program is even started with little to no configuration. Therefore, the sending program can be completely divorced from the receiver in terms of dependencies. In addition, UDS is agnostic towards payload size and can pause the sending program until the FIFO queue of data has begun to move and there is memory available to continue sending.

We further clarify the operation of Pallex through the scenario presented in Figure 4.12. In this example, it is assumed that the user application is composed of two processes (or threads), and the first process depends on the data generated by the second process to complete its task. Also, each process can be broken into a set of sequentially running stages. For the stages we assume that,

$$\mathrm{D}(s_i) = \emptyset,\ \ \mathrm{D}(s_j) = \{s_i\},\ \ \mathrm{D}(s_k) = \{s_i, s_j\},$$

$$\mathrm{D}(s_m) = \emptyset,\ \ \mathrm{D}(s_n) = \{s_m\},\ \ \mathrm{D}(s_l) = \{s_i, s_j, s_k, s_m, s_n\}$$

Each stage also depends on a set of units. Figure 4.12 shows the instances where the stage dependency set and unit dependency set of each stage are resolved. Since $\mathrm{D}(s_i) = \{\}$ and $\mathrm{D}(s_m) = \{\}$, both stages can start at time t_1 at which point their unit dependency sets are resolved. Therefore, these two stages run in parallel while the userspace units are being activated. At time t_2, stage s_i completes. However, s_j cannot be started because its unit dependency set is resolved at t_3. Therefore, s_i is blocked and waits until stage s_j is started and is ready to receive the data generated by s_i. Sharing the messages generated by a stage s_i with the next stages is denoted as M_{s_i}. This figure also shows that, since $\mathrm{D}(s_l) = \{s_i, s_j, s_k, s_m, s_n\}$, stage s_l cannot be started before the completion of s_n. Once started, stage s_l reads M_{s_n} and M_{s_k}, the messages shared by two stages s_n and s_k. At time t_9, both the unit and stage dependency sets of s_l are resolved and this stage has all the resources necessary to complete its operation. The user application finishes at time t_{10}. At this point the system enters the shutdown phase.

The performance improvements achieved by Pallex depend on the decomposition of user application into stages. In order to minimize the waiting time of each stage, it is important to break a user application into stages with smallest unit dependency sets. However, given the large number of units activated during P_{usi}, finding the right decomposition might not be a straightforward task. In addition, little to no improvement is observed by minimizing the waiting time of user application stages on units where their execution time is only a few milliseconds. Due to the sampling-processing-sending nature of IoT applications, we can narrow down the list of important units to simplify the task of decomposition, as follows.

– GPIO. These interfaces are initialized by the GPU. Therefore, as soon as `systemd` finishes its initialization (0.75s for the RPi3 and 1.2s for the RPiZW), user applications can use the GPIO pins. In addition to enabling the execution of user applications at the beginning of P_{usi}, using GPIOs provides a faster communication interface without the high overhead of a full networking stack, especially when a small amount of data must be communicated between nearby devices.

- I2C, SPI, CSI. The drivers for I2C (Inter-Integrated Circuit), SPI (Serial Peripheral Interface) and CSI (Camera Serial Interface) are loaded as kernel modules during P_{knl}. Therefore, the unit dependency of user applications relying on these components is resolved at the beginning of P_{usi}.

- Bluetooth. Bluetooth depends on both `hciuart.service` and `bluetooth.service`. As Section 4.3 showed, the activation of these services finishes after `rc.local`. Therefore, user application stages that rely on Bluetooth must be started by creating a `systemd` service that starts after `bluetooth.service` instead of `rc.local`. If an application includes tasks that do not depend on these services, then running those tasks as stages that start before the completion of these services can result in a considerable performance improvement, especially due to their long activation duration.

- Ethernet. Similar to WiFi, the status of the Ethernet interface can be derived from `network.target`. A difference between the WiFi and Ethernet interfaces is that the speed of initializing Ethernet is faster because it does not perform the authentication and association process that is required for WiFi. Therefore, using Ethernet results in a shorter duty cycle. However, since most IoT applications rely on wireless communications, we mainly focus on WiFi in this paper.

- WiFi. Stages that rely on WiFi must be initialized after the activation of `network.target`. Another option is to start the stage after `network-`

`online.target`, which is invoked once the network is *connected* as opposed to *available*.

In Appendix A.1 we present an overview of units to provide the users with guidelines regarding the impact of each unit on each application scenario.

4.5.2 User Application Scenarios for Evaluating Pallex

In this section, we evaluate the performance of Pallex when applied to various types of user applications implemented on the RPi3 and RPiZW. Since using the RPi3 or RPiZW is justified when the application at hand cannot be accomplished using resource-constrained devices (such as those employing ARM Cortex-M or R processor), our user application scenarios include heavy operations such as image capture, encryption and classification. However, it should be noted that we omit image classification using RPiZW due to the high overhead caused by running the machine learning algorithm on this platform. We explain these scenarios in the following subsections.

Scenario 1: Image Capture (IC)

For this scenario we used a camera module [80] to capture an image. The application stage set includes only one stage, $\mathrm{P} = \{s_{cap}\}$, where $\mathrm{D}(s_{cap}) = \emptyset$ and $\mathrm{D}'(s_{cap}) = \emptyset$. Therefore, we use unit configuration EU for this scenario. The camera module is able to capture one picture per second (each around 2.5MB) and uses CSI to communicate with the RPi. After capturing the image, the image is saved as a JPEG file by the user application.

Scenario 2: Image Capture+Encryption (IC&E)

We extend "Scenario 1" by encrypting the captured image using the AES-256 encryption algorithm. The application stage set includes one stage, P = $\{s_{cap+enc}\}$, where D($s_{cap+enc}$) = $\emptyset$ and D′($s_{cap+enc}$) = $\emptyset$. Please note that since encryption depends on capture, we do not decompose the application into two stages. Unit configuration EU is used for this scenario.

Scenario 3: Image Capture+Classification (IC&C)

In this scenario, we capture an image and classify it using a pre-trained K-Nearest Neighbors (KNN) algorithm [86]. Since image capture and loading the model are independent, we decompose the application into three stages: image capture (s_{cap}), loading the KNN model (s_{load}), and performing classification (s_{clas}). Therefore, P = $\{s_{cap}, s_{load}, s_{clas}\}$, where D($s_{cap}$) = $\emptyset$, D′(s_{cap}) = $\emptyset$, D(s_{load}) = $\emptyset$, D′(s_{load}) = $\emptyset$, D(s_{clas}) = $\{s_{cap}, s_{load}\}$ and D′(s_{cap}) = $\emptyset$. Please note that s_{cap} and s_{load} are two concurrently running processes. Consequently, although the two heavy stages are image capture and loading the model, both stages start and run concurrently. s_{clas} depends on the completion of s_{cap} and s_{load} to perform its operation. Unit configuration EU is used for this scenario.

Scenario 4: Image Capture+Upload (IC&U)

In this scenario, we capture an image and transmit it to a cloud server through WiFi communication with an access point. The application stage set includes two stages, P = $\{s_{cap}, s_{upl}\}$, where D(s_{cap}) = $\emptyset$, D′(s_{cap}) = $\emptyset$, D(s_{upl}) = $\{s_{cap}\}$ and D′(s_{upl}) = $\{u_{net}\}$, where u_{net} refers to the `network-online.target` unit. Please note that the service configuration we used for this scenario is EU w/NET1.

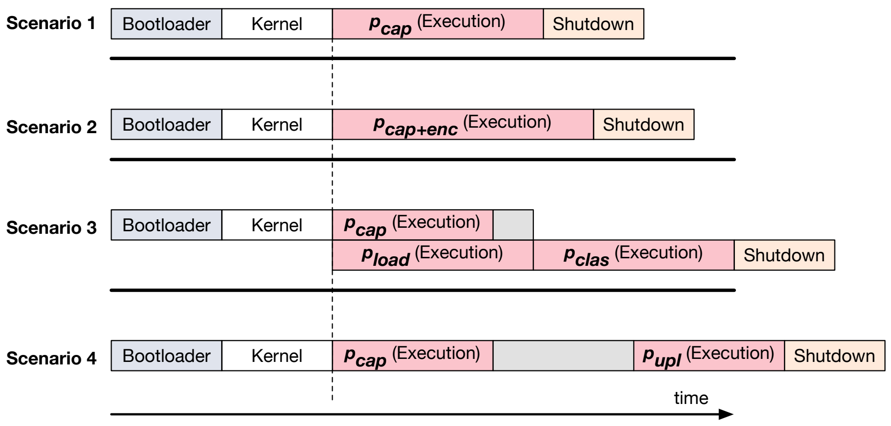


Figure 4.13: The four scenarios used to measure the effect of Pallex on duration and energy consumption. We have used three variations of Scenario 4 to evaluate Pallex when using WiFi for uploading data.

Because networking services require a relatively long duration to initialize, we are interested in evaluating Pallex in scenarios where T_{usi} is long. For the RPi3 and RPiZW, we vary the number of images that are captured and uploaded from 1 to 3. For the RPi3, activating `networking.service` is approximately 3.5s, which means that we can capture a maximum of 3 images in parallel with this service activation without negatively impacting the duration of userspace initialization. Although activating `networking.service` on the RPiZW is longer than 7.4s, we cap the maximum number of images at 3 to present a fair comparison across the two boards. These sub-scenarios are referred to as IC&Ux, where x refers to the number of images captured and uploaded.

Figure 4.13 shows a summary of the operations of these applications versus time.

4.5.3 Results

Considering the user application scenarios given in Section 4.5.2, in this section we present the performance measurement results of Pallex. It must be noted that

Pallex is compared against baseline scenarios which employ unit configuration to prevent the activation of unnecessary units. In the baseline scenarios, referred to as "normal" in the figures, the user application is started when `rc.local` is loaded. Therefore, if unit configuration was not applied to the baselines, their duration and energy consumption would be higher.

In terms of duration and energy measurement, we consider the interval between the instance the RPi is powered on until the completion of the user application. Please note that energy measurement is particularly important because the OS dynamically adjusts the operating frequency of the processor cores based on load [87] (a.k.a., dynamic frequency scaling). Hence, it is important to verify that the additional system load during P_{usi} does not eliminate the energy saving achieved by reducing the duration.

Figure 4.14 and 4.15 show the performance improvements achieved using Pallex for RPi3 and RPiZW during the boot up process, respectively. In these figures, the black and gray bars represent P_{btl} and P_{knl}, respectively. It must be noted that, since Pallex affects system performance after P_{knl}, both E_{btl} and E_{knl} are fixed irrespective to the RPi board used. From the userspace processing point of view, for the IC scenario, energy consumption is reduced by 24.7% and 10.77% for the RPi3 and RPiZW, respectively. For the IC&E scenario, we observe 27.22% and 5.98% improvement in terms of duration and 21.54% and 4.89% in terms of energy for the RPi3 and RPiZW, respectively. Both of these scenarios include a single stage that is executed as soon as `systemd` is ready. It must be noted that the user application is not the only process running after the completion of `systemd` initialization. As explained in Section 4.3.2, the units that are essential for maintaining system integrity and stability are being activated during this duration as well.

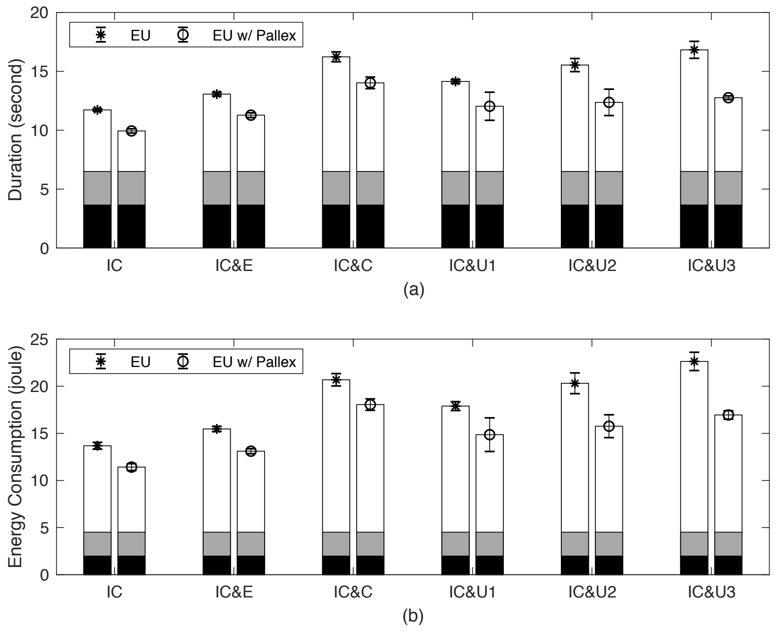


Figure 4.14: The duration and energy consumption of Pallex versus normal launching of user applications. The device used is a RPi3. Black, grey, and white bars show the bootloader phase, kernel phase, and the completion of user application.

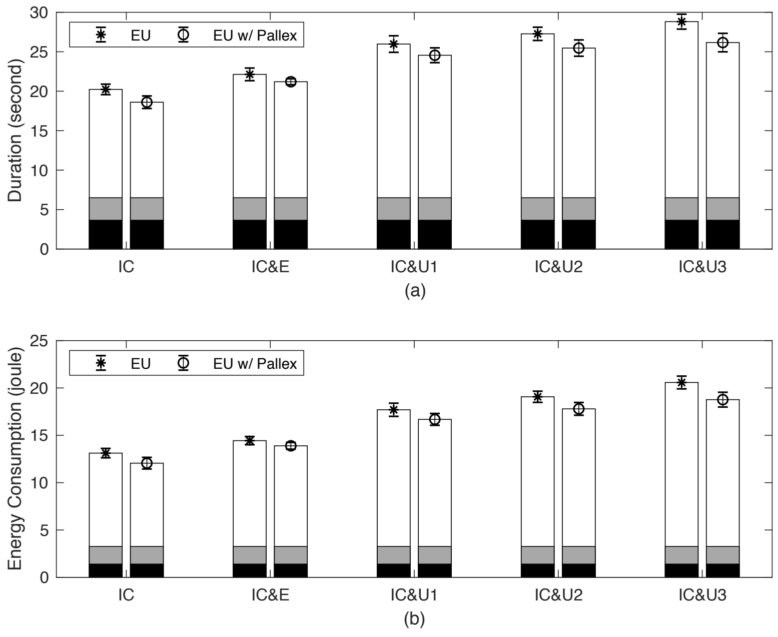


Figure 4.15: The duration and energy consumption of Pallex versus normal launching of user applications. The device used is a RPiZW. Black, grey, and white bars show the bootloader phase, kernel phase, and the completion of user application.

Each IC&C scenario is composed of three stages, where two stages run in parallel and during P_{usi}. For this scenario we observe a 22.68% improvement in terms of duration and 16.29% in terms of energy for the RPi3.

The IC&U scenario is composed of two stages, where the first stage runs during P_{usi}. As we showed in the previous sections, activating the `networking.service` is a lengthy process due to the initialization of hardware and networking utilities as well as association with the access point. Therefore, this scenario can significantly benefit from capturing images while activating networking services is in progress. In addition, more improvement is observed as the number of captured images increases: When one image is captured, we observe 27.57% and 7.3% improvement in terms of userspace processing duration for the RPi3 and RPiZW, respectively. These improvements are increased to 39.36% and 11.89% for these two boards when three images are captured. In terms of energy improvement, we observe 31.35% and 10.45% improvement for the RPi3 and RPiZW, respectively, when three images are captured.

Comparing the two hardware platforms, we can observe that when Pallex is applied, the RPi3 shows a higher performance improvement compared to the RPiZW. These results are consistent with our observations in Section 4.3.2, indicating that the RPi3 platform consumes less energy than the RPiZW in a duty-cycling capacity despite drawing more current. The RPi3 parallelizes userspace initialization processes across multiple cores, resulting in a shorter duty-cycle duration. The impact of shortening the processing duration is greater than the impact of the difference in current consumption across platforms.

In order to measure the impact of Pallex on the lifetime of duty-cycled sys-

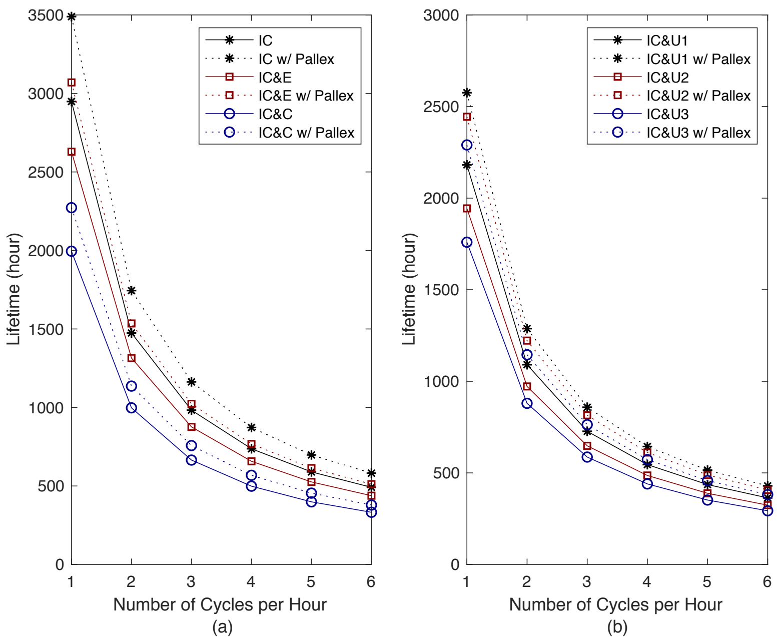


Figure 4.16: Lifetime of the RPi3 for different user application scenarios. Sub-figure (a) shows networking-independent scenarios, and sub-figure (b) shows networking-dependent scenarios.

tems, we compute system lifetime as follows:

$$\begin{aligned} \text{lifetime} &= \frac{E_{bat}}{(E_{btl} + E_{knl} + E_{usr} + E_{sdn}) \times N} \\ &= \frac{3600 \times 2400 \times 10^{-3} \times 5}{(E_{btl} + E_{knl} + E_{user} + E_{sdn}) \times N} \end{aligned} \tag{4.3}$$

where E_{btl}, E_{knl}, E_{user}, and E_{sdn} are the energy consumption of the bootloader phase, kernel phase, user application, and shutdown phase. E_{bat} is the available energy of the battery, and N is the number of cycles per hour. For the shutdown phase, we used the *forced shutdown* mechanism detailed in Section 4.4, as it is faster than the `shutdown` command without sacrificing reliability. We also assume the capacity of the battery is 2400mAh and its voltage is 5V. Figures 4.16 and 4.17 show system lifetime versus the number of cycles per hour for the RPi3 and RPiZW, respectively.

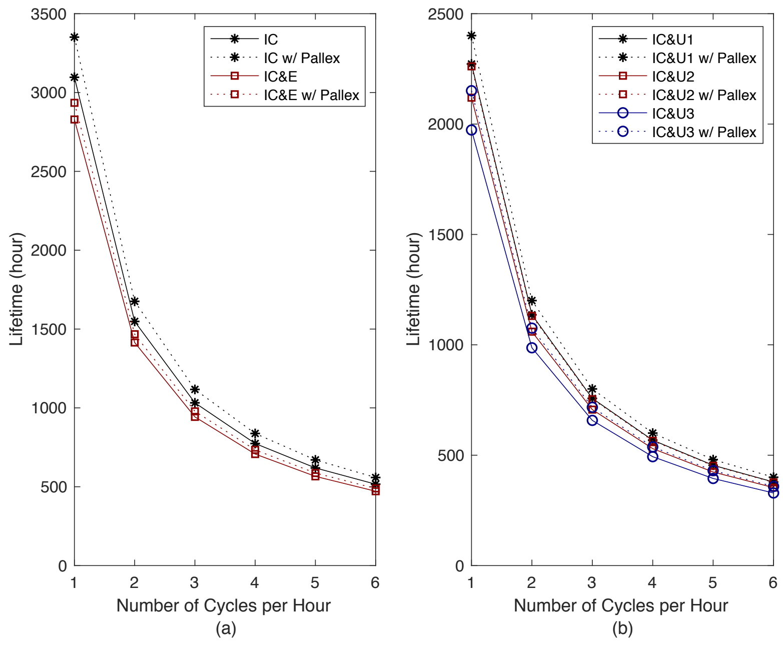


Figure 4.17: Lifetime of the RPiZW for different user application scenarios. Sub-figure (a) shows networking-independent scenarios, and sub-figure (b) shows networking-dependent scenarios.

The maximum improvement in lifetime for the RPi3 is 30.16% for scenario IC&U3, and the minimum is 13.89% for scenario IC&C. Similarly, for the RPiZW, the maximum improvement is 9.01% for scenario IC&U3, and the minimum is 3.74% for scenario IC&C. For the networking-dependent scenarios, we notice that lifetime improvement increases with respect to the number of images captured and transmitted. Specifically, for the RPi3, we notice an improvement of 18.06% for IC&U1, 25.74% for IC&U2, and 30.16% for IC&U3. For the RPiZW, we notice an improvement of 5.67% for IC&U1, 6.7% for IC&U2, and 9.01% for IC&U3. For the networking-independent scenarios, IC%C achieves the highest increase in lifetime (18.33%) on the RPi3. This is attributed to the higher processing demand and concurrent execution of s_{cap} and s_{load} stages.

The improvements in lifetime, in particular, reduce the cost of energy harvesting systems as well as system maintenance. For example, when the amount of energy consumed per hour is reduced, a lower-cost energy harvesting system (e.g., smaller solar panels, smaller batteries) can be used as it is provided with more time to harvest and store energy. For fully battery powered systems, increasing lifetime reduces the frequency of system maintenance to replace or recharge the batteries.

Chapter 5 Conclusion

In this thesis we tackled two main challenges related to the energy efficiency of IoT systems. First, we designed and developed an energy measurement platform designed to address the scalability and accuracy issues of measuring IoT devices. Second, we studied and improved energy consumption in Linux based IoT devices through optimizing boot time, shutdown time, and user application time.

To solve the various challenges relating to energy measurement, we developed EMPIOT, a low-cost, programmable, and accurate energy measurement solution that can operate effectively on a wide range of IoT devices. By analyzing the performance of different drivers, we have determined the BCM driver results in higher sampling rate and lower energy consumption. In addition we were able to conclude that lowering the shield's operational voltage slightly reduces sampling rate by experimenting with the voltage modes. By experimenting with different data structures and algorithms for data aggregation, we conclude that using two buffers and batching file writes results in lower energy consumption. To prove its versatility we evaluated EMPIOT's performance using five different IoT devices and four types of workloads. Our results confirm an energy measurement error less than 3% which proves that our platform can work for different kinds of IoT devices. In addition to setting our goals for building the energy measurement platform, we also provide valuable insights into designing Linux-based IoT systems which use the I2C bus. We provide this through our study of the I2C performance and the drivers that interface with it. Finally, we believe have developed a platform which satisfies the demanding requirements for power profiling on IoT devices. By achieving accuracy with five different IoT devices with different settings we have developed a platform which will generalize well with multiple kinds of devices. By experimenting with data structures, I2C drivers, and calibration we improved performance. Finally through a rich feature set which utilizes various peripherals on the base board, we are able to make EMPIOT easy to integrate with existing devices and testbeds.

The increasing number of Linux-based IoT devices used for edge and fog computing necessitates the adoption of duty-cycling mechanisms to reduce the energy consumption of these devices. To this end, profiling and improving the operation of userspace initialization offers techniques that can be easily adopted by users. In this work, we presented a thorough study of the Linux boot up process, in particular the effects of unit activation on the duration and energy consumption of the boot up and shutdown phases. We showed that although some units cannot be disabled without compromising system stability, duty-cycling performance is significantly enhanced by application-specific unit configuration. Our studies also showed that there is no effect on boot up performance when up to 95% of the SDC capacity is utilized. However, using a faster SDC results in a slightly shorter, more energy efficient duty-cycle. After analyzing the resource utilization of the RPi3 and RPiZW during the boot up process, we showed that user applications can be executed in parallel with the userspace initialization phase to reduce the energy consumption of each duty-cycle. We did so by proposing Pallex, a parallel execution framework which relies on `systemd` and Unix Domain Sockets to break and execute a user application into multiple phases. Our evaluations show up to a 31% reduction in energy consumption and up to a 30% enhancement in lifetime when Pallex is applied to various IoT application scenarios. Our studies also reveal the trade-off between processing power and current consumption. Although the current draw of the RPiZW is lower than that of the RPi3, this paper confirms the RPi3 platform is more suitable for duty-cycled applications. This is because the RPi3 parallelizes userspace initialization and user application processes across multiple cores, resulting in lower energy consumption by shortening processing duration.

Although we were able to achieve our goals with EMPIOT, there is still more that can be done to study and improve the platform. Some of the future work avenues are as follows: Although the energy consumption of simple wireless technologies such as 802.15.4 and LoRa have been thoroughly studied and modeled, in addition to 802.11b/n standards, newer technologies, such as 802.11ac, NB-IoT, and eMTC, are being used for IoT applications. The EMPIOT platform provides a low-cost and scalable solution to deploy testbeds and profile the energy consumption of these complex wireless technologies. Another area of future work is to port EMPIOT to non-Linux-based systems and measure its sampling rate and overhead when the operating system of the base board is a RTOS. Finally we determined that the measurement range of EMPIOT makes it a suitable energy measurement platform for other applications, such as measuring and evaluating the effect of energy efficiency techniques proposed for cloud computing platforms [88, 89, 90].

We also noticed that there are certain areas where we can improve boot up time and shutdown time on Linux-based devices. Some potential areas of future work are as follows: Although the studies of this paper revealed the significant effects of quad-core and single-core SoCs on duty-cycling performance, extending these observations and profiling the performance of other COTS Linux-based boards is of interest. From the shutdown point of view, developing a model to predict the probability of SDC corruption based on factors such as capacity, I/O rate, and duty-cycling frequency enables users to choose the best shutdown mechanism available without compromising system reliability. Regarding Pallex, although we have provided guidelines to simplify its applicability to other IoT scenarios, it would be helpful to develop a program to analyze user application code and break it into stages based on the tasks it performs and the dependencies of those tasks. Finally, unit configuration and Pallex can be integrated with bootloader and kernel-level optimization mechanisms to further enhance performance.

Appendix A
Appendix

A.1 List of Units in Raspbian Stretch Lite (RSL)

In this section, we present an overview of the units available in RSL. All units except those in the EU category can be disabled if they are not necessary for the application scenario being considered.

A.1.1 Essential Units (EU)

– `boot.mount`: This unit helps `systemd` resolve dependency trees for units that depend on mounting `/boot` before activation.

– `dev-mmcblk0p2.device`: This unit brings the root partition on the SDC into the scope of `systemd` so that units that require the root partition's mount to finish before activation can resolve their dependencies properly.

– `dev-mqueue.mount`: This unit informs `systemd` when the POSIX message queues for internal system messages is ready.

– `kmod.service`: This service contains `modprobe`, which is used for loading and unloading kernel modules.

– `kmod-static-nodes.service`: This service creates a list of required static modules for the loaded kernel.

- `run-rpc_pipefs.mount`: This unit directs `systemd` on how to mount the RAM-based `pipefs`, which is used every time a process is forked or a pipe ("|") is used.
- `sys-kernel-debug.mount`: Similar to `dev-mmcblk0p2.device`, this unit helps `systemd` resolve dependencies correctly. The actual mounting of `debugfs` occurs within `udev`, which is the daemon that detects hardware changes.
- `sys-kernel-config.mount`: This unit prevents the system from reaching `sysinit.target` until the kernel configuration parameters are fully loaded into the kernel from the Configuration File System (`configfs`).
- `systemd-fsck.service` and `systemd-fsck-root.service`: These services run `fsck` on each partition to ensure file system consistency. This is an important step, and does not run every time unless there are problems detected on the SDC.
- `systemd-journald.service`: Many programs rely on `journald` for logging output, including the kernel (through `kmsg`). Therefore, it should not be disabled. However, in order to speed up its initialization, it may be useful to lower the size limit of the journal logs since a dependency, `systemd-journal-flush.service`, must rotate this log file on initialization.
- `systemd-modules-load.service`: This service starts early in the userspace initialization phase to load static kernel modules.
- `systemd-remount-fs.service`: In the beginning of the userspace initialization phase, this service mounts the necessary API filesystems for the kernel (such as `/proc`, `/sys`, or `/dev`) to a RAM disk.

- `systemd-random-seed.service`: This service loads the random seed and saves it at shutdown to enable the device to generate a new value when the system restarts.

- `systemd-sysctl.service`: By loading kernel configurations, this service enables `systemctl` to perform as expected.

- `systemd-udevd.service`: This service initializes `udev`, a daemon that listens to kernel `uevents` and matches them against specified rules, to run scripts. For example, it can load drivers when a new device is attached, or mount a USB drive when it is plugged in.

- `systemd-udev-trigger.service`: Devices plugged in before the system is powered on might not generate the kernel messages necessary for `udev` to discover them. This service probes and detects devices that `udev` would not normally discover.

- `sudo.service`: This service clears cached sudo privilege escalations to enforce user re-authentication after every reboot.

- `systemd-tmpfiles-setup.service` and `systemd-tmpfiles-setup-dev.service`: Mount `/tmp` and delete the old files. These services also create any files that are specified by user-provided configuration.

- `systemd-rfkill.service`: This service restores the `rfkill` state at the beginning of userspace initialization to ensure it matches the status saved before shutdown. Therefore, if the wireless peripherals (typically WiFi or Bluetooth) had `rfkill` preventing their use before shutdown, they will remain disabled on reboot.

- `systemd-update-utmp.service` and `systemd-update-utmp-runlevel.service`: Record and manage the system uptime, the logged-in users, and users' log-in method (such as ssh, tty, and serial port).
- `systemd-user-sessions.service`: This service enables user log-in and denies log-in attempts after the shutdown signal has been sent. If a device in the field does not require log-in capabilities, this service does not need to be started automatically. For example, it can be started by a helper program when a GPIO pin is pulled high.
- `systemd-logind.service`: This service is responsible for tasks such as user session management, processor usage quotas, and device access management.

A.1.2 Networking-related Services (NRS)

- `avahi-daemon.service`: This service enables programs to discover and publish services and hosts running on a local network. Note that this service can significantly slow the speed of the `ifdown` command and therefore the shutdown process if not completely uninstalled. Unless necessary for the user application, `avahi-daemon` should be uninstalled.
- `bluetoothd.service`: Daemon for controlling the Bluetooth interface. `Bluez`, `bluetoothctl`, and many other Bluetooth-related utilities communicate through this daemon.
- `dhcpcd.service`: The daemon responsible for managing the DHCP protocol on all targeted network interfaces. This service can be disabled if the device does not require a network connection or is guaranteed a static IP address.

The duration of IP allocation depends on external factors including link quality and the load of access point.

– `hciuart.service`: This is responsible for initializing the HCI bluetooth interface. HCI stands for "Host-to-Controller-Interface" and it is controlled over a serial UART interface.

– `networking.service`: Completes the configuration of WiFi and Ethernet interfaces based on the settings available in the `/etc/network/interfaces` configuration file.

– `nfs-config.service`: This service, along with `nfs-common.service`, loads configuration details applicable to Network File Systems (NFS).

– `rsyncd.service`: Daemon that listens on port 873 for incoming `rsync` file transfer requests. `rsync` is used for efficiently transferring and synchronizing files across computer systems.

– `rpcbind.service`: This service accepts requests for Remote Procedure Calls (RPC) and binds them to TCP ports for access and control.

– `sshd.service`: This service belongs to the `OpenSSH` package. It runs in the background to listen for and accept or deny incoming ssh connections according to a user-defined configuration file.

– `systemd-hostnamed.service`: This service can be used to control the host-name and related metadata by user programs.

– `systemd-networkd.service`: Brings up the system's network manager and provides it with discovered networks, both physical and virtual.

– `systemd-resolved.service`: Provides local DNS resolution for namespaces such as `localhost` or those added by the user to overlay DNS provided by an external source.

– `systemd-timesyncd.service`: Used for time synchronization across the network.

A.1.3 Memory Management Services

– `dphys-swapfile.service`: This service initializes, mounts, unmounts, and deletes swap files on the SDC. If the available RAM is enough for the user application, then disabling this service results in a performance enhancement in terms of faster boot up time and prolonged SDC lifetime. This service is usually required when the user application involves loading large machine learning models and data sets.

A.1.4 I/O-related Services

– `alsa-utils.service`: Represents the tools relating to the Advanced Linux Sound Architecture (ALSA).

– `alsa-restore.service`: Initializes and restores the last state of the RPi's onboard soundcard.

A.1.5 Miscellaneous Units

– `fake-hwclock.service`: This service saves the current time to a file at shutdown and loads it at boot up time. Without this service, the RPi is unaware of the current epoch time until it establishes a network connection.

An incorrect epoch value may cause some files to appear as if they are edited in the future.

- `plymouth.service`: Provides a flicker-free graphical boot up process. Other related services include `plymouth-quit.service`, `plymouth-quit-wait.service`, `plymouth-start. service`, and `plymouth-read-write.service`.

- `raspi-config.service`: This service loads configuration changes made by the user such as processor governance, display overscan, and filesystem partition expansions, and applies them on reboot.

- `rsyslog.service`: Tools for log processing and conversion.

- `console-setup.service`: Configures the fonts, screen resolution, keyboard layout, etc., for virtual tty terminals.

Bibliography

[1] Gartner. IoT Units Installed Base by Category (Millions of Units). [Online]. Available: https://www.gartner.com/newsroom/id/3598917 1

[2] J. Hong, Y.-G. Hong, and J.-S. Youn, "Problem Statement of IoT integrated with Edge Computing," 2018. [Online]. Available: https://tools.ietf.org/html/draft-hong-iot-edge-computing-00 2

[3] E. Griffiths, S. Assana, and K. Whitehouse, "Privacy-preserving image processing with binocular thermal cameras," *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, vol. 1, no. 4, pp. 133:1–133:25, 2018. 2

[4] R. Kelly, "Internet of things data to top 1.6 zettabytes by 2022," *Campus Technology*, vol. 9, pp. 1536–1233, 2016. 2

[5] B. Dezfouli, M. Radi, and O. Chipara, "Rewimo: a real-time and reliable low-power wireless mobile network," *ACM Transactions on Sensor Networks (TOSN)*, vol. 13, no. 3, p. 17, 2017. 2, 7

[6] M. Chiang and T. Zhang, "Fog and IoT: An overview of research opportunities," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 854–864, 2016. 2

[7] P. Delforge, "Slashing energy use in computers and monitors while protecting our wallets, health, and planet." Natural Resources Defense Council, 2016. 2

[8] S. Chu and A. Majumdar, “Opportunities and challenges for a sustainable energy future,” *nature*, vol. 488, no. 7411, p. 294, 2012. 2

[9] Z. Wang, Y. Liu, Y. Sun, Y. Li, D. Zhang, and H. Yang, “An energy-efficient heterogeneous dual-core processor for Internet of Things,” in *IEEE International Symposium on Circuits and Systems (ISCAS).* IEEE, 2015, pp. 2301–2304. 2

[10] M. Ueki, K. Takeuchi, T. Yamamoto, A. Tanabe, N. Ikarashi, M. Saitoh, T. Nagumo, H. Sunamura, M. Narihiro, K. Uejima *et al.*, “Low-power embedded reram technology for iot applications,” in *Symposium on VLSI Technology.* IEEE, 2015, pp. T108–T109. 2

[11] Cypress Semiconductor. CYW43907: IEEE 802.11 a/b/g/n SoC with an Embedded Applications Processor. [Online]. Available: http://www.cypress.com/file/298236/download 2, 13, 29

[12] E. Baccelli, O. Hahm, M. Gunes, M. Wahlisch, and T. C. Schmidt, “RIOT OS: Towards an OS for the Internet of Things,” in *IEEE Conference on Computer Communications Workshops (INFOCOM Workshops).* IEEE, 2013, pp. 79–80. 2

[13] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer *et al.*, “TinyOS: An operating system for sensor networks,” in *Ambient intelligence.* Springer, 2005, pp. 115–148. 2

[14] (2018) ThreadX RTOS Real-Time Operating System. [Online]. Available: https://rtos.com/solutions/threadx/real-time-operating-system/ 2, 7

[15] (2018) The FreeRTOS Kernel. [Online]. Available: https://www.freertos.org 2, 7

[16] C. E. Jones, K. M. Sivalingam, P. Agrawal, and J. C. Chen, "A survey of energy efficient network protocols for wireless networks," *wireless networks*, vol. 7, no. 4, pp. 343–358, 2001. 2

[17] B. Dezfouli, M. Radi, K. Whitehouse, S. A. Razak, and T. Hwee-Pink, "DICSA: Distributed and concurrent link scheduling algorithm for data gathering in wireless sensor networks," *Ad Hoc Networks*, vol. 25, pp. 54–71, 2015. 2, 7

[18] S. Zoican and M. Vochin, "LwIP stack protocol for embedded sensors network," in *9th International Conference on Communications.* IEEE, 2012, pp. 221–224. 2

[19] F. Kaup, P. Gottschling, and D. Hausheer, "PowerPi: Measuring and modeling the power consumption of the Raspberry Pi," in *Proceedings of the 39th Annual IEEE Conference on Local Computer Networks (LCN'14)*, 2014, pp. 236–243. 3, 12, 34

[20] K. Gomez, R. Riggio, T. Rasheed, D. Miorandi, and F. Granelli, "Energino: a Hardware and Software Solution for Energy Consumption Monitoring," in *Proceedings of the International Workshop on Wireless Network Measurements (WiOpt'12)*, 2012, pp. 311 – 317. 3, 15, 34

[21] R. Zhou and G. Xing, "Nemo: A high-fidelity noninvasive power meter system for wireless sensor networks," *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN'13)*, pp. 141–152, 2013. 3, 4, 14, 34

[22] T. Trathnigg, M. Jürgen, and R. Weiss, “A low-cost energy measurement setup and improving the accuracy of energy simulators for wireless sensor networks,” in *Proceedings of the workshop on Real-world wireless sensor networks*, 2008, pp. 31–35. 3, 4, 15, 34, 36

[23] J. Eriksson, F. Österlind, N. Finne, A. Dunkels, N. Tsiftes, and T. Voigt, “Accurate network-scale power profiling for sensor network simulators.” in *Proceedings of the International Conference on Embedded Wireless Systems and Networks (EWSN’09)*, vol. 9. Springer, 2009, pp. 312–326. 3, 34

[24] T. Stathopoulos, D. McIntire, and W. J. Kaiser, “The energy endoscope: Real-time detailed energy accounting for wireless sensor nodes,” *Proceedings of International Conference on Information Processing in Sensor Networks (IPSN’08)*, pp. 383–394, 2008. 3, 13

[25] Q. Li, M. Martins, O. Gnawali, and R. Fonseca, “On the effectiveness of energy metering on every node,” in *Proceedings of the IEEE International Conference on Distributed Computing in Sensor Systems (DCoSS’13)*, 2013, pp. 231–240. 3

[26] B. Dezfouli, M. Radi, S. A. Razak, T. Hwee-Pink, and K. A. Bakar, “Modeling low-power wireless communications,” *Journal of Network and Computer Applications*, vol. 51, pp. 102–126, 2015. 3, 7

[27] B. Dezfouli, M. Radi, K. Whitehouse, S. A. Razak, and H.-P. Tan, “Cama: Efficient modeling of the capture effect for low-power wireless networks,” *ACM Transactions on Sensor Networks (TOSN)*, vol. 11, no. 1, p. 20, 2014. 3

[28] Keysight Technologies. U8000 Series, Single Output DC Power Supplies. [Online]. Available: https://literature.cdn.keysight.com/litweb/pdf/5989-7182EN.pdf?id=1460471 3, 31

[29] Tektronix. DMM7510 7.5-Digit Graphical Sampling Multimeter. [Online]. Available: https://www.tek.com/tektronix-and-keithley-digital-multimeter/dmm7510 3, 31, 37

[30] I. Haratcherev, G. Halkes, and T. Parker, "PowerBench: A Scalable Testbed Infrastructure for Benchmarking Power Consumption," in *Proceedings of the International Workshop on Sensor Network Engineering (IWSNE'08)*, 2008, pp. 37–44. 4, 14, 36

[31] J. Andersen and M. T. Hansen, "Energy Bucket: A Tool for Power Profiling and Debugging of Sensor Nodes," in *Proceedings of Third International Conference on Sensor Technologies and Applications (SENSORCOMM'09)*. IEEE, 2009, pp. 132–138. 4, 13

[32] X. Jiang, P. Dutta, D. Culler, and I. Stoica, "Micro Power Meter for Energy Monitoring of Wireless Sensor Networks at Scale," *Proceedings of the 6th international conference on Information processing in sensor networks (IPSN'07)*, p. 186, 2007. 4, 13

[33] P. Dutta, M. Feldmeier, J. Paradiso, and D. Culler, "Energy metering for free: Augmenting switching regulators for real-time monitoring," in *Proceedings of the 7th International Conference on Information Processing in Sensor Networks (IPSN'08)*, 2008, pp. 283–294. 4, 13, 36

[34] INA219 Zero-Drift, Bidirectional Current/Power Monitor With I2C Interface. [Online]. Available: http://www.ti.com/lit/ds/symlink/ina219.pdf 4, 23

[35] BCM4343W: 802.11b/g/n WLAN, Bluetooth and BLE SoC Module. [Online]. Available: https://products.avnet.com/opasdata/d120001/medias/docus/138/AES-BCM4343W-M1-G_data_sheet_v2_3.pdf 5

[36] "Linux I2C Driver." [Online]. Available: https://www.kernel.org/doc/Documentation/i2c/dev-interface 5

[37] Eclipse Foundation, "Key Trends from the IoT Developer Survey 2018." [Online]. Available: https://blogs.eclipse.org/post/benjamin-cabe/key-trends-iot-developer-survey-2018 7, 11

[38] F. Kaup, P. Gottschling, and D. Hausheer, "PowerPi: Measuring and modeling the power consumption of the Raspberry Pi," in *39th Conference on Local Computer Networks (LCN).* IEEE, 2014, pp. 236–243. 7

[39] R. Morabito, "Virtualization on internet of things edge devices with container technologies: a performance evaluation," *IEEE Access*, vol. 5, pp. 8835–8850, 2017. 7

[40] V. Vujovic and M. Maksimovic, "Raspberry Pi as a Wireless Sensor node: Performances and constraints," in *37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO).* IEEE, 2014, pp. 1013–1018. 7

[41] R. Fisher, L. Ledwaba, G. Hancke, and C. Kruger, "Open hardware: A role to play in wireless sensor networks?" *Sensors*, vol. 15, no. 3, pp. 6818–6844, 2015. 7

[42] K. H. Chung, M. S. Choi, and K. S. Ahn, "A study on the packaging for fast boot-up time in the embedded Linux," in *13th International Conference*

on Embedded and Real-Time Computing Systems and Applications (RTCSA). IEEE, 2007, pp. 89–94. 8, 20, 21

[43] C. Villegas. (2006) Improve the Debian boot process. [Online]. Available: http://bootdebian.blogspot.com 8

[44] H. Kaminaga, "Improving linux startup time using software resume (and other techniques)," in *Linux Symposium*, 2006, p. 17. 8, 19

[45] C. Villegas and P. Reinholdtsen, "State-of the-art in the boot process." Google Summer of Code, 2006. [Online]. Available: https://pdfs.semanticscholar.org/a171/696ddb41ba8aad53cdcbb6aba1c4547aa80e.pdf 8, 16, 17

[46] T. R. Bird, "Methods to Improve Bootup Time in Linux," in *Proceedings of the Linux Symposium*, 2004. 8, 18, 20, 21

[47] G. Singh, K. Bipin, and R. Dhawan, "Optimizing the boot time of android on embedded system," in *15th International Symposium on Consumer Electronics (ISCE).* IEEE, 2011, pp. 503–508. 8, 9, 17

[48] H. Jo, H. Kim, J. Jeong, J. Lee, and S. Maeng, "Optimizing the startup time of embedded systems: a case study of digital tv," *IEEE Transactions on Consumer Electronics*, vol. 55, no. 4, 2009. 8, 9, 17

[49] L. M. Feeney and M. Nilsson, "Investigating the energy consumption of a wireless network interface in an ad hoc networking environment," in *Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'01)*, 2001, pp. 1548–1557. 12

[50] A. Milenkovic, M. Milenkovic, E. Jovanov, D. Hite, and D. Raskovic, "An environment for runtime power monitoring of wireless sensor network plat-

forms," in *Proceedings of the Thirty-Seventh Southeastern Symposium on System Theory (SSST'05)*, 2005, pp. 406–410. 12

[51] Measurement Computing Corporation. USB-1608FS-Plus data acquisition device. [Online]. Available: http://www.mccdaq.com/pdfs/manuals/USB-1608FS-Plus.pdf 13

[52] BQ2019: Advanced Battery Monitor. [Online]. Available: https://www.ti.com/lit/ds/slus465e/slus465e.pdf 13

[53] CC2650 SimpleLink Multistandard Wireless MCU. [Online]. Available: https://www.ti.com/lit/ds/swrs158b/swrs158b.pdf 13

[54] S. Naderiparizi, A. N. Parks, F. S. Parizi, and J. R. Smith, "μMonitor: In-situ energy monitoring with microwatt power consumption," in *Proceedings of the IEEE International Conference on RFID (RFID'16)*. IEEE, may 2016, pp. 1–8. 14, 36

[55] A. Pötsch, A. Berger, and A. Springer, "Efficient analysis of power consumption behaviour of embedded wireless iot systems," in *Instrumentation and Measurement Technology Conference (I2MTC)*, 2017, pp. 1–6. 14, 36

[56] A. Pötsch, A. Berger, C. Leitner, and A. Springer, "A power measurement system for accurate energy profiling of embedded wireless systems," in *Proceedings of the 19th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'14)*, 2014, pp. 1–4. 14

[57] N. Zhu and I. O'Connor, "Energy measurements and evaluations on high data rate and ultra low power wsn node," in *10th IEEE International Conference on Networking, Sensing and Control (ICNSC)*, 2013, pp. 232–236. 15

[58] R. Hartung, U. Kulau, and L. Wolf, "Distributed energy measurement in WSNs for outdoor applications," *Proceedings of the 13th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON'16)*, 2016. 15

[59] S. Keranidis, G. Kazdaridis, V. Passas, T. Korakis, I. Koutsopoulos, and L. Tassiulas, "NITOS Energy Monitoring Framework: Real Time Power Monitoring in Experimental Wireless Network Deployments," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 18, no. 1, pp. 64–74, 2014. 16

[60] Cramfs: cram a filesystem onto a small ROM. [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/cramfs.txt 18

[61] G. Lim, J. young Hwang, K. Park, and S.-B. Suh, "Enhancing init scheme for improving bootup time in mobile devices," *2015 Eighth International Conference on Mobile Computing and Ubiquitous Networking (ICMU)*, pp. 149–154, 2015. 18

[62] C. Park, K. Kim, Y. Jang, and K. Hyun, "Linux bootup time reduction for digital still camera," in *Linux Symposium*, 2006, p. 231. 19

[63] "BCM2835 ARM Peripherals." [Online]. Available: https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf 19, 49

[64] I. Joe and S. C. Lee, "Bootup time improvement for embedded linux using snapshot images created on boot time," in *The 2nd International Conference on Next Generation Information Technology (ICNIT).* IEEE, 2011, pp. 193–196. 19

[65] strace: Linux syscall tracer. [Online]. Available: https://strace.io 28

[66] R. Lim, F. Ferrari, M. Zimmerling, C. Walser, P. Sommer, and J. Beutel, "FlockLab: A Testbed for Distributed, Synchronized Tracing and Profiling of Wireless Embedded Systems," in *Proceedings of the 12th international conference on Information processing in sensor networks (IPSN'13)*, 2013, p. 153. 36

[67] Analog Devices. 256-Position and 33-Position Digital Potentiometers. [Online]. Available: http://www.analog.com/media/en/technical-documentation/data-sheets/AD5200_5201.pdf 37

[68] "Adafruit INA219 Current Sensor Breakout." [Online]. Available: https://learn.adafruit.com/adafruit-ina219-current-sensor-breakout/downloads 38

[69] E. Upton, J. Duntemann, R. Roberts, B. Everard, and T. Mamtora, *Learning Computer Architecture with Raspberry Pi.* John Wiley & Sons, 2016. 49

[70] (2018) systemd System and Service Manager. [Online]. Available: https://www.freedesktop.org/wiki/Software/systemd/ 50

[71] J. Gorauskas, "Managing Services in Linux: Past, Present and Future," *Linux J.*, vol. 2015, no. 251, 2015. 50

[72] "systemd.unit," 2018. [Online]. Available: https://www.freedesktop.org/software/systemd/man/systemd.unit.html 50

[73] R. Love, *Linux Kernel Development*, 3rd ed. Addison-Wesley, 2010. 51

[74] B. Dezfouli, I. Amirtharaj, and C.-C. Li, "EMPIOT: An energy measurement platform for wireless IoT devices," *Journal of Network and Computer Applications*, vol. 121, pp. 135 – 148, 2018. 51

[75] "C library for Broadcom BCM 2835." [Online]. Available: http://www.airspayce.com/mikem/bcm2835/ 52

[76] "Wiring Pi: GPIO Interface library for the Raspberry Pi." [Online]. Available: http://wiringpi.com 52

[77] "systemctl," 2017. [Online]. Available: https://www.freedesktop.org/software/systemd/man/systemctl.html 59, 72

[78] "The GNU Awk User's Guide." [Online]. Available: https://www.gnu.org/software/gawk/manual/gawk.html 63

[79] S. Godard, "iostat," 2018. [Online]. Available: http://man7.org/linux/man-pages/man1/iostat.1.html 63

[80] (2018) Camera Module (v2). [Online]. Available: https://www.raspberrypi.org/documentation/hardware/camera/ 65, 79

[81] (2018) IMX219PQ: Diagonal 4.6mm 8.08M-Effective Pixel Color CMOS Image Sensor. [Online]. Available: https://www.sony-semicon.co.jp/products_en/new_pro/april_2014/imx219_e.html 65

[82] S. K. Paul Rubin, David MacKenzie, "dd: convert and copy a file," 2018. [Online]. Available: http://man7.org/linux/man-pages/man1/dd.1.html 66

[83] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems.* ACM, 2009, pp. 229–240. 69

[84] "Linux Programmer's Manual, Socket," 2018. [Online]. Available: http://man7.org/linux/man-pages/man2/socket.2.html 75

[85] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: from I/O ports to process management.* " O'Reilly Media, Inc.", 2005. 75

[86] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992. 80

[87] D. Brodowski, N. Golde, R. J. Wysocki, and V. Kumar, "CPU frequency and voltage scaling code in the Linux (TM) kernel." [Online]. Available: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt 82

[88] B. Aldawsari, T. Baker, and D. England, "Trusted energy efficient cloud-based services brokerage platform," *Int. J. Intell. Comput. Res*, vol. 6, pp. 630–639, 2015. 90

[89] T. Baker, Y. Ngoko, R. Tolosana-Calasanz, O. F. Rana, and M. Randles, "Energy efficient cloud computing environment via autonomic meta-director framework," in *Sixth International Conference on Developments in eSystems Engineering (DeSE).* IEEE, 2013, pp. 198–203. 90

[90] T. Baker, M. Asim, H. Tawfik, B. Aldawsari, and R. Buyya, "An energy-aware service composition algorithm for multiple cloud-based iot applications," *Journal of Network and Computer Applications*, vol. 89, pp. 96–108, 2017. 90