Title

by
Kurt Gu

Professor Kelly Shaw, Advisor

A thesis submitted in partial fulfillment
of the requirements for the
Degree of Bachelor of Arts with Honors
in Computer Science

Williams College
Williamstown, Massachusetts

April 11, 2023

Contents

List of Figures

List of Tables

Abstract

Acknowledgments

Chapter 1

Introduction

1.1 Motivation

1.2 Summary

Chapter 2

Background

2.1 Summary

Chapter 3

Methodology

In this thesis, we will be implementing a small energy-harvesting device using physical hardware, on which we will run two software applications and evaluate how our proposed mitigations affect their energy efficiency performances. This chapter introduces the specific hardware and software components used to build our device, the applications that will run on the device, and the testing methodology we use to determine the energy efficiency of those applications.

3.1 Hardware Tools

3.1.1 System Overview

Our test device consists of four major parts:

1. The microcontroller unit (MCU) connects to all other periphral components, and handles data processing and computation.
2. The power supply, which itself contains three parts: the supercapacitor that serves as the battery for the device, the supercapacitor manager that uses incoming harvested power to charge the supercapacitor, and finally the energy harvester such as a solar panel. In our case, we use a variable power supply in place of an energy harvester for better consistency and stability between different experiments.
3. The camera module, which captures image sensory data.
4. The long range radio, or LoRa, which transmits the data captured and processed by our device to an external and distant receiver.

Figure 3.1 is a diagram of how the components in our device are connected to each other.

3.1.2 Component Selection

1. MCU: For our device, we select the MSP430FR5994 low-power MCU from Texas Instruments (TI). This is an MCU with 8kB of RAM, 256kB of nonvolatile memory, and a clock frequency

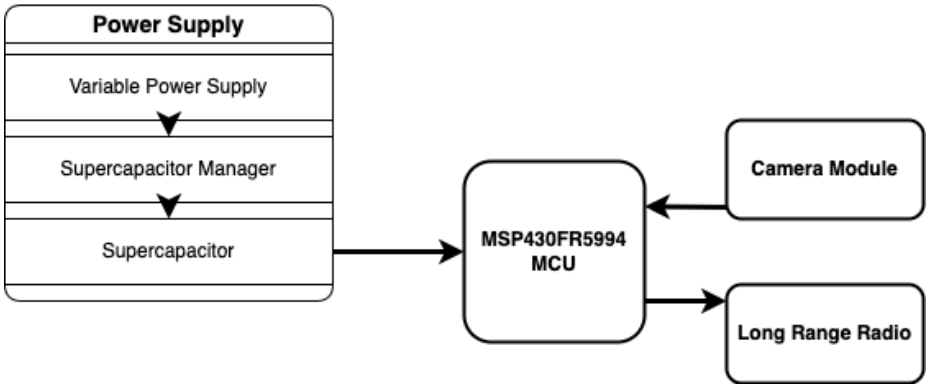

Figure 3.1: Diagram of component connections.

range between 10kHz to 24MHz. It also carries an onboard temperature sensor. Specifically, we are using the LaunchPad development kit for this MCU, which comes with headers for the MCU's general purpose IO pins for easy connection to periphrals, and includes an onboard debug probe for programming, debugging and energy measurements. The dev kit is also equipped with a microSD card slot, and a small onboard supercapacitor battery.

2. Power supply: We select the AEMSUCA [tindie source] based on the AEM10941 chip as our supercapacitor manager for its wide compatibility and low cost. For the supercapacitor, we select a 1F, 2.7V capacitor from Kemet[source], which achieves a good balance between capacity and size.

3. Camera module: We select the low-power HM01B0 camera module, which is capable of capturing images up to 320x320 pixles in size.

4. Long range radio: We select the RFM95W long range radio transceiver. It has a transmission range of approximately 2 kilometers or 1.24 miles, and operates in either the 868 or 915 MHz channel.

3.1.3 Component Power Consumption

Since the focus of this thesis is optimizing energy efficiency, we should pay attention to hardware power consumption. Therefore, in table 3.1.3 we include a list of power draws for each of our components. Note that the power consumption of the MCU varies under different workloads. It typically increases as the workload becomes more computationally intensive, and as the clock frequency increases.

3.2 Software Tools

In this thesis, we write software to test the performance of our proposed efficiency mitigations. The main saftware tasks required to perform the tests are as follows:

Component	State	Power Draw (mW)
MCU	Low Power(10-32 kHz clock, periphals off)	0.34
MCU	Normal(1 MHz clock)	1.22
MCU	High Frequency (24 MHz clock)	2.2
Camera	Standby	< 0.2
Camera	Active (160x120 pixels @ 30 fps)	< 1.1
Camera	Active (320x320 pixels @ 30 fps)	< 2
Radio	Standby	6
Radio	Transmit	66 - 396

Table 3.1: Component power draws [sources]

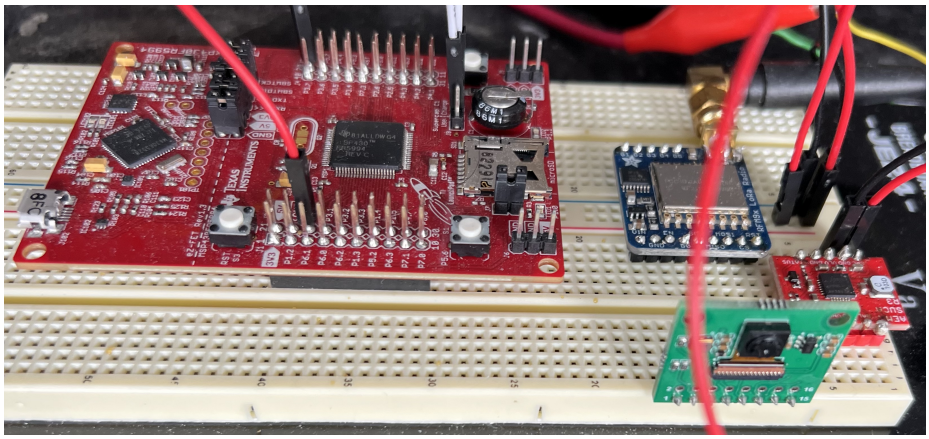

Figure 3.2: Our test device, with each component labeled.

1. Load data from SD card for local inference, or poll onboard sensor for temperature data.
2. Process the captured data to determine if it is an event of interest.
3. Communicate with the long range radio chip to transmit interesting data.
4. Based on the running task, adjust clock frequency to either save power or boost performance.
5. Execute all the tasks above according to a user-specified schedule.

In this section, we introduce the software tools used to implement the tasks above.

3.2.1 Software Environment

To program the device, we use TI's own integrated development environment (IDE) called Code Composer Studio (CCS) [source]. Source code is written in C, and CCS compiles our C code using MSP-CGT, which is TI's code generation tool for compiling device-specific assembly code. CCS then load the compiled code onto our device through the onboard debugger via a USB connection. CCS's code editor also provides advice on best practices for writing energy efficient C code.

Package Name	Description
MSP430 General Library	Standard register and bit definitions for MSP430 microcontrollers, included with Code Composer Studio.
MSP430 DriverLib [cite]	Periphal driver library, allows application development at an API level, reduces register-level programming.
MSP430 DSPLib [cite]	Digital Signal Processing library, contains optimized functions for vector/matrix operations using fixed-point arithmetics.
Lenet-accelerator [cite]	LeNet-5 image classification network, adpated to the MSP430FR5994 architecture using DSPLib.
SDCardLib [cite]	SD card driver library specifically for our development kit, enables writing to and reading from files on an inserted SD card.

Table 3.2: Packages

3.2.2 Development Process

3.2.3 External Packages

Working with embedded MCUs such as the MSP430 series often involve programming directly at the register level, which requires writing complex code with low readbility. For a quicker and simpler development process, we use a number of external packages developed by TI and the open-source community. For example, the MSP430 DriverLib provides methods that alter the MCU's clock frequency, so that the programmer no longer needs to manually unlock and modify the registers controlling clock speeds. Table 3.2 provides a list of packages we used during software implementation, along with a short description of their usage.

3.3 Software Applications

In this thesis, we focus on optimizing the energy efficiency for two specific applications: inference using a local classification network, and recording environment temperature using an onboard sensor. This section introduces general structure of an application running on energy harvesting devices, as well as the two specific applications we focus on, while Section 3.4 discusses the efficiency mitigations we propose and test.

3.3.1 Application Structure

3.3.2 Application: Local Inference

Limitations Of Local Inference (Move to Experiments section?)

As we have discussed previously, running machine learning models locally to classify interesting data reduces the frequency of energy intensive data transmissions. However, preforming local inference also comes with limitations, primarily in terms of model size and performance. Since our MCU has only 8kB of RAM and 256kB of nonvolatile memory, larger models would not fit in memory. Furthermore, since our MCU cannot compute floating point numbers natively, local inference models using

floating-point weights yield non-ideal performance when run on our device without optimizations.

LeNet-5 Classification Network

For this thesis, we use local inference to classify image data as either interesting or uninteresting. Specifically, we select the LeNet-5 image classification network[source] as our local inference model. First published in 1998 by Yann LeCun et al, LeNet-5 is a relatively small neural network with seven total layers, and a reasonable number of weights that can fit onto our device's memory, which makes it an ideal candidate for our use. Since LeNet-5 has already been adapted to the MSP430 platform using native libraries and fixed-point arithmetics[source], it performs well when run on our test device. Furthermore, LeNet-5 has been trained and tested on many available datasets, which makes its on-device performance easier to evaluate without compiling our own dataset.

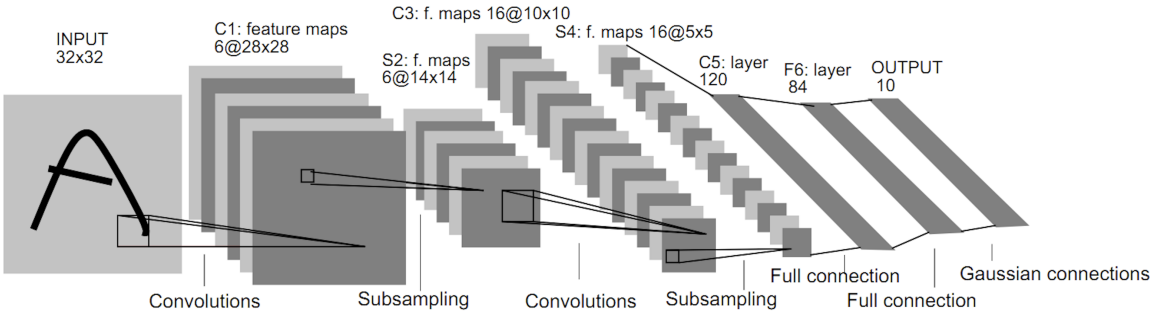

Figure 3.3: Structure of LeNet-5.[source]

Datasets Used For Evaluation

We use existing datasets to evaluate the performance of our local inference model. Specifically, we have identified three datasets that can be used with LeNet:

1. MNIST: The Modified National Institute of Standards and Technology (MNIST) database is a collection of handwritten digits$(0-9)$. Each example in the dataset is a 28×28 greyscale image.

2. Fashion-MNIST: A dataset containing 28×28 grayscale images of fashion products from 10 categories, with $7,000$ images in each category.

3. CIFAR-10: A dataset containing 32×32 color images from 10 categories, with $6,000$ images in each category.

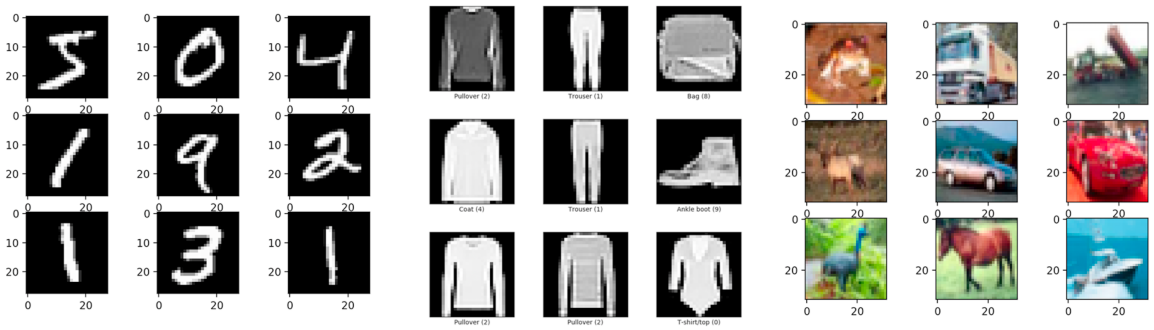

Figure 3.4: Sample images from datasets used to test ML model performance.
Left: MNIST, Middle: Fashion-MNIST, Right: CIFAR-10

3.3.3 Application: Temperature Sensing With Sprinting

Data Accquisition

Tunable Parameters

3.4 Testing Methodology

3.4.1 Supplying Power & Data

As mentioned in section 3.1.1, power is supplied to our test device via a variable power supply, charging the supercapacitor which then powers the test device. By using a variable power supply, we can alter the magnitude of output power to simulate the behavior of a solar panel under different lighting conditions. To simulate a cloudy day with low harvested power, we could lower the power supply's output current. To simulate a sunny day, we could simply raise the output current.

For our temperature sensing application, our device generates data in real-time using the on-device sensor. However, to evaluate the performance of the local inference application, we need to provide our device with test datasets containing hundreds of images, which are too large to fit onto the 256kB of local memory. Therefore, we make use of the onboard SD card slot by loading the test datasets onto an microSD card, from which the MCU will then read during testing. The drawback to this approach is that the SD card requires around 40mW of additional power. However, since this power draw is near-constant, it can be treated as a bias in our data and can be factored out easily without affecting data accuracy.

3.4.2 Monitoring Power With Energytrace

Energytrace is a power monitoring tool integrated in CCS that works with our development kit. It monitors the device during operation using the debugger onboard the development kit, and provides the programmer with in-depth information about the device's power consumption behavior.

As an example, Figure 3.5 contains two sample outputs by Energytrace. The Energytrace++ profiler at the top records energy consumption and runtime information about each power mode that our device could be in, and about the program being executed on-device, while the basic Energytrace profiler at the bottom records the device's electrical information during operation, such as voltage, current, average and max power draw, etc.

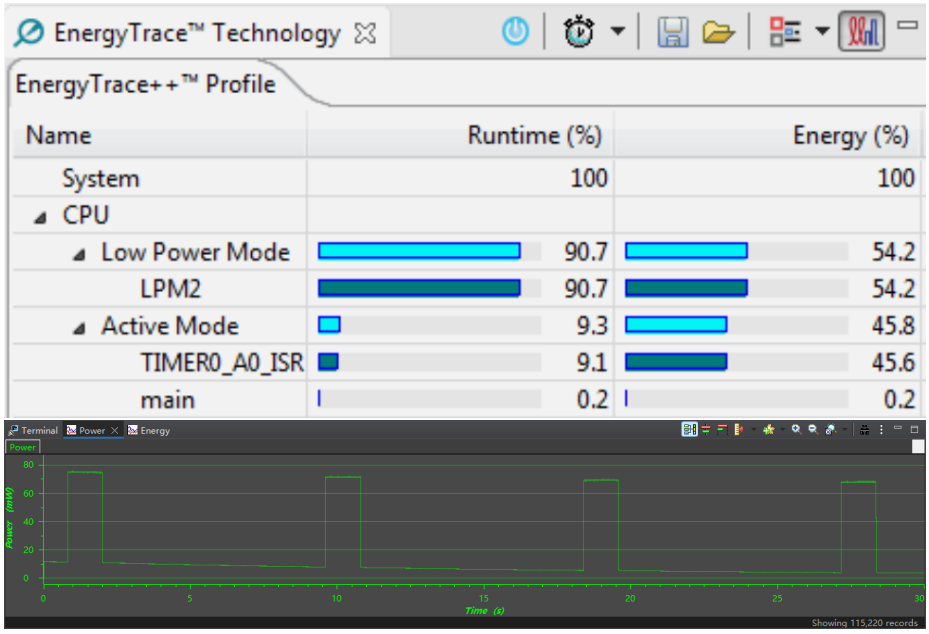

Figure 3.5: Example Energytrace outputs.

3.4.3 General Testing Framework (Move to experiments?)

General Structure

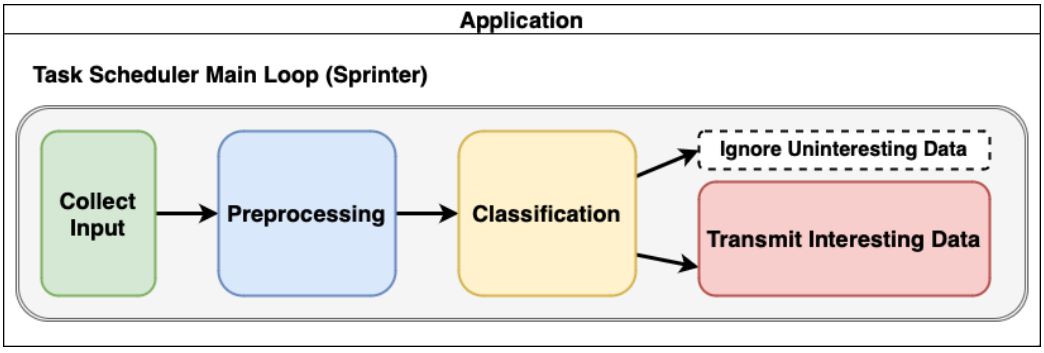

Figure 3.6: Structure of an EH Device Application.

Preprocessing Methods

Clock Frequency Manipulation

Work Schedule Manipulation

3.4.4 Evaluating Performance (Efficiency Improvements)

Ideally, the performace and energy efficiency of our test device should be evaluated using a real-world workload in an energy harvesting scenario. If our mitigations achieve better performance or efficiency than the existing baselines, we can argue that they constitute an improvement to the system. However, in this thesis we opt for a more simplistic approach: under stable power (constant power input from power supply, or power from a fully charged supercapacitor, etc.), if our mitigations reduce the average power consumption of the device overall compared to the baseline results, then we can reasonably argue that our mitigations constitute an improvement in energy efficiency. This approach also uses existing datasets, fed to the device sequentially over time to simulate the occurance of events of interest.

Our approach is roughly equivalent to an ideal deployment scenario with an even distribution of interesting events and sufficient harvested energy (battery always full or always turn on at full charge), and serves as a reasonable approximation of real-world scenarios, while removing the additional complexity from environmental factors such as weather variability and occurance frequency of interesting events.

3.5 Summary

In this chapter we introduced the hardware components of our image and temperature sensing test system, built around Texas Instruments' MSP430 MCU. We also introduced the software tools we use to build applications, run tasks and tests, and evaluate the performance of our purposed mitigations. We discussed the two applications that this thesis will focus on: local inference for image data, and computational sprinting with temperature data. Finally, we provided details on our testing methodology when running experiments. Specifically, we talked about how power and data are supplied to the test system, how power usage can be monitored in detail through Energytrace, how experiments will be performed to test our purposed mitigations, and lastly, how we evaluate their performance in terms of energy efficiency.

Chapter 4

Experiments

4.1 Baseline

4.1.1 Sense-and-Send Baseline

4.1.2 Local Inference Baseline

4.2 Testing Preprocessing Methods

4.2.1 Implementation

4.3 Testing Sprinting Strategies

4.3.1 Implementation

4.4 Summary

Chapter 5

Results

5.1 Summary

Chapter 6

Conclusion

6.1 Summary