5-2016

Efficient and Reliable Task Scheduling, Network Reprogramming, and Data Storage for Wireless Sensor Networks

Biswajit Mazumder
Clemson University, bmazumd@g.clemson.edu

Efficient and Reliable Task Scheduling, Network Reprogramming, and Data Storage for Wireless Sensor Networks

---

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

---

by
Biswajit Mazumder
May 2016

Accepted by:
Dr. Brian Malloy, Committee Chair
Dr. Jason O. Hallstrom, Committee Co-Chair
Dr. Robert Geist
Dr. Jacob Sorber

Abstract

Wireless sensor networks (WSNs) typically consist of a large number of resource-constrained nodes. The limited computational resources afforded by these nodes present unique development challenges. In this dissertation, we consider three such challenges.

The first challenge focuses on minimizing energy usage in WSNs through intelligent duty cycling. Limited energy resources dictate the design of many embedded applications, causing such systems to be composed of small, modular tasks, scheduled periodically. In this model, each embedded device wakes, executes a task-set, and returns to sleep. These systems spend most of their time in a state of deep sleep to minimize power consumption. We refer to these systems as *almost-always-sleeping* (AAS) systems. We describe a series of task schedulers for AAS systems designed to maximize sleep time. We consider four scheduler designs, model their performance, and present detailed performance analysis results under varying load conditions.

The second challenge focuses on a fast and reliable network reprogramming solution for WSNs based on incremental code updates. We first present *VSPIN*, a framework for developing incremental code update mechanisms to support efficient reprogramming of WSNs. VSPIN provides a modular testing platform on the host system to "plug-in" and evaluate various incremental code update algorithms. The framework supports *Avrdude*, among the most popular Linux-based programming tools for AVR microcontrollers. Using VSPIN, we next present an incremental code update strategy to efficiently reprogram wireless sensor nodes. We adapt a linear space and quadratic time algorithm (*Hirschberg's Algorithm*) for computing *maximal common subsequences* to build an *edit map* specifying an edit sequence required to transform the code running in a sensor network to a new code image. We then present a heuristic-based optimization strategy for efficient edit script encoding to reduce the edit map size. Finally, we present experimental results exploring the reduction in data size that it enables. The approach achieves reductions of 99.987% for simple changes,

and between 86.95% and 94.58% for more complex changes, compared to full image transmissions — leading to significantly lower energy costs for wireless sensor network reprogramming.

The third challenge focuses on enabling fast and reliable data storage in wireless sensor systems. A file storage system that is fast, lightweight, and reliable across device failures is important to safeguard the data that these devices record. A fast and efficient file system enables sensed data to be sampled and stored quickly and batched for later transmission. A reliable file system allows seamless operation without disruptions due to hardware, software, or other unforeseen failures. While flash technology provides persistent storage by itself, it has limitations that prevent it from being used in mission-critical deployment scenarios. Hybrid memory models which utilize newer non-volatile memory technologies, such as ferroelectric RAM (FRAM), can mitigate the physical disadvantages of flash. In this vein, we present the design and implementation of LoggerFS, a fast, lightweight, and reliable file system for wireless sensor networks, which uses a hybrid memory design consisting of RAM, FRAM, and flash. LoggerFS is engineered to provide fast data storage, have a small memory footprint, and provide data reliability across system failures. LoggerFS adapts a log-structured file system approach, augmented with data persistence and reliability guarantees. A caching mechanism allows for flash wear-leveling and fast data buffering. We present a performance evaluation of LoggerFS using a prototypical in-situ sensing platform and demonstrate between 50% and 800% improvements for various workloads using the FRAM write-back cache over the implementation without the cache.

Dedication

I would like to dedicate this work to my wife, Dr. Sritama Nath, and my parents, Dr. Gayatri Mazumder and Mr. Bijan Mazumder.

Acknowledgments

There are many people who I would like to thank during my years at Clemson University. First and foremost, I would like to thank my advisor, Dr. Jason O. Hallstrom, for being a friend, philosopher, and guide, and for his continued support, patience, and understanding during the past six years. Second, I would like to thank my committee members, Dr. Brian A. Malloy, Dr. Robert M. Geist, and Dr. Jacob M. Sorber, for the help and suggestions they provided on my research that led to this dissertation. Third, I would like to thank my extended DSRG lab family — Hao, Jiannan, Sally, Yvon, and Yang for their friendship, help, and moral support over the years. Fourth, I would like to thank all my friends at Clemson, who have been a constant source of joy and inspiration. Last but not the least, I wish to thank my family. I would like to thank my wife, Dr. Sritama Nath, and my parents, Dr. Gayatri Mazumder and Mr. Bijan Mazumder, for always believing in me and being the source of my strength and happiness. This would not have been possible without your support! Thank you all!

Table of Contents

Title Page . . . . . **i**

Abstract . . . . . **ii**

Dedication . . . . . **iv**

Acknowledgments . . . . . **v**

List of Tables . . . . . **viii**

List of Figures . . . . . **ix**

List of Listings . . . . . **x**

1 Introduction **1**
1.1 Problem Statement 2
1.2 Research Approach and Contributions 5
1.3 Dissertation Organization 9

2 Background **10**
2.1 Task Scheduling 10
2.2 Network Reprogramming 11
2.3 Data Storage 14

3 Task Scheduling **17**
3.1 Basic Scheduler 18
3.2 $O(1)$ Scheduler 20
3.3 $O(n)$ Scheduler 21
3.4 Intelligent Sleep Scheduler 22
3.5 Algebraic Models 25
3.6 Evaluation 28
3.7 Summary 32

4 Network Reprogramming **34**
4.1 VSPIN Framework 34
4.2 Incremental Code Update Mechanism 41
4.3 Evaluation 50
4.4 Summary 55

5 Data Storage **59**
5.1 LoggerFS Design 59
5.2 File System Implementation 66

5.3 Evaluation 70
5.4 Summary 76

6 Related Work 77
6.1 Task Scheduling 77
6.2 Network Reprogramming 78
6.3 Data Storage 81

7 Conclusion 84
7.1 Contribution Summary 84
7.2 Expected Impact 86

Bibliography 86

List of Tables

1.1 Memory Technology Comparison 7

4.1 Energy Consumption Characteristics 52
4.2 Edit Map Sizes for C-based OS and TinyOS Update Scenarios 53
4.3 Edit Map Sizes for Different Merge Windows in Cases 1-5 54
4.4 Comparison of Simple and Incremental Reprogramming Costs 55

5.1 Measured Read and Write Speeds 72

List of Figures

2.1 A_1, A_2, A_{TASK}, ω_i, and N 11

3.1 ISR Execution Profile ($\mathtt{t_{ISR}}$) (Intelligent Sleep Scheduler) 27
3.2 X = $\frac{n_2}{n_1+n_2}$, Y = $\frac{\mathtt{n_{task\_executed}}}{\mathtt{n_{in\_queue}}}$, Z = $n_1A_1 + n_2A_2$ 29
3.3 *Null Activation Period* Contributions (X = $\frac{n_2}{n_1+n_2}$, Y = n_1A_1 μs) 30
3.4 Scheduler Power Consumption Profiles 31
3.5 Battery Life Expectancy 32

4.1 VSPIN Architecture 35
4.2 VSPIN Kernel Module 36
4.3 Linux *tty* Modules and VSPIN Kernel Module Interaction 37
4.4 VSPIN User Process 38
4.5 Incremental Code Update Process 40
4.6 Incremental Network Reprogramming Workflow 41
4.7 Logical View of the LCS Between Two Images 43
4.8 *Edit Map* Generation Flowchart 44
4.9 Update and Data Node Structures 45
4.10 Update Ordering Problem 46
4.11 Edit Map Encoding Scheme 47
4.12 Application of Edit Operations 48
4.13 Effects of *Map Optimization* on $\mathtt{C_{update}}$ 57
4.14 Reprogramming Costs for Different Code Update Scenarios 58

5.1 LoggerFS Hardware Architecture 60
5.2 Hybrid Storage Approach 60
5.3 Logical View of File and Record Layout 61
5.4 File Metadata in RAM and FRAM 62
5.5 Consistent Data Structure Update 63
5.6 Read Path with Cache 65
5.7 Write Path with Cache 66
5.8 Consistent File Metadata 69
5.9 Write Throughput with Cache 73
5.10 Read Throughput with Cache 74
5.11 50%/50% Read-Write Throughput 75

List of Listings

2.1 *Avrdude* usage example 12
3.1 `scheduler_run`() (Basic Scheduler) 18
3.2 `scheduler_run`() and `run_task`() ($O(1)$ Scheduler) 19
3.3 `scheduler_run`() ($O(n)$ Scheduler) 21
3.4 `scheduler_run`() and `intelligent_sleep`() (Intelligent Sleep Scheduler) 23
3.5 Overflow ISR (Intelligent Sleep Scheduler) 24
5.1 File, Record, and Cache Metadata 67
5.2 LoggerFS API 68

Chapter 1

Introduction

Wireless sensor networks (WSNs) usually consist of a large number of nodes [2] and are used in a wide variety of applications, such as disaster response, volcanic activity monitoring, structural health monitoring, environmental monitoring, medical monitoring, and traffic monitoring [15, 34, 41, 42, 46, 63, 64, 67, 68, 70]. Due to the nature of these applications, the WSN nodes are often required to operate for extended periods of time without any human intervention. To enable longer operation times (and lower costs), the nodes are typically resource-constrained, thus consuming less power. The scarcity of computational resources presents WSN developers with unique challenges. The wide range of associated applications, along with the desired scale of the associated deployments further exacerbates the design challenges. In this dissertation, we focus on three key system design challenges for WSNs – task scheduling, network reprogramming, and persistent data storage.

Task Scheduling: A significant class of embedded applications are characterized by low duty-cycle operation and time-triggered, periodic execution. Task scheduling and duty-cycling in such applications presents a fundamentally different scheduling paradigm than witnessed in resource-abundant systems; traditional task scheduling strategies are not optimal.

Network Reprogramming: The ability to remotely reprogram nodes is essential in installing and maintaining large WSN deployments. Adding new functionality or addressing defects in existing applications after a deployment is complete is often not feasible due to time and personnel constraints. Many state-of-the-art solutions do not support fast and efficient network reprogramming.

Data Storage: Sensor applications that support high data rate sampling require WSN nodes to store this data in a fast and efficient manner. Mission critical applications also require fault-tolerant data storage. Current storage technologies for embedded sensor devices do not provide a sufficiently robust solution; they either lack the ability to support fast sampling rates, or are unable to provide persistence guarantees in the presence of faults.

1.1 Problem Statement

In this section, we describe the challenges to be addressed in this dissertation in the contexts of task scheduling, network reprogramming, and data storage in sensor networks.

1.1.1 Task Scheduling

A significant class of embedded applications are characterized by low duty-cycle operation and time-triggered, periodic execution. These systems sleep for relatively long periods, wake in response to an interrupt, perform a short computation, and return to sleep. We refer to these systems as *almost-always-sleeping* (AAS) systems. The wireless sensor network domain is rife with representative examples. Environmental monitoring networks [63, 64, 42], for instance, comprise distributed sensors that periodically wake to collect and transmit environmental stimuli before returning to sleep. Indeed, *most* sensing systems – environmental or otherwise – adopt a variant of this strategy, as do numerous other embedded applications.

The broad adoption of AAS designs is due to the energy efficiency they afford. Modern microcontrollers support sleep states in which internal circuitry may be powered-down, reducing energy consumption by several orders of magnitude. As an example, common wireless sensor networking platforms consume 10s of *milli*watts in the active state, and only 10s of *micro*watts when idle [50]. For devices that exhibit this two-phase consumption profile, the best conservation strategy is to sleep as often and as long as possible.

The active period of an embedded device is partitioned into two components: the time spent executing application code (*tasks*), and the time spent executing scheduling code. Reducing the runtime of individual tasks can only be achieved on an application-by-application basis. **Reducing the scheduling overhead, however, can be achieved through careful analysis and design of the underlying scheduling system — our focus**.

1.1.2 Network Reprogramming

WSNs typically consist of resource-constrained sensor nodes to enable low power consumption and longer operation times. The applications are developed and compiled on typical desktop systems, and then written to the flash program memory of target nodes using an in-system-programmer (ISP), or other serial reprogramming approach [44, 35]. These approaches handle only one sensor node at a time, causing network programming time to increase linearly with network size. Many WSNs, e.g. for disaster management, structural health monitoring, and volcanic activity monitoring [15, 34, 67], may require a large number of nodes to be deployed, rendering these programming approaches unusable.

The ability to reprogram nodes is also essential in maintaining large WSN deployments. If new functionality must be added, or changes must be made to correct defects after a large network has been deployed, the in-system-programming and serial reprogramming approaches can be prohibitively time-consuming and costly. Network reprogramming using wireless communication to transfer program images to sensor nodes mitigates the problems posed by these approaches. While wireless methods of data dissemination enable sensor devices freedom from direct connections to the host system, wireless data transmission is energy-intensive. Previous studies have reported that transmission of a single bit of data requires 1,000 times the energy required for the execution of a single instruction on typical devices [51, 56, 71]. Brute force update mechanisms which use wireless communication to reprogram nodes are not energy efficient, as they require the entire code image to be transmitted throughout the network. Incremental code update strategies significantly reduce the amount of data that must be transferred to the reprogramming system, thus improving the energy footprint of the network. **Integrating incremental update strategies can result in faster and more efficient network reprogramming — our primary focus**.

Many incremental code update strategies have been proposed [31, 32, 51, 45] to reduce the amount of time required for reprogramming. However, very little information has been reported in the literature regarding the supporting software engineering tools and frameworks used to develop these reprogramming strategies. As a result, the design of every new incremental update mechanism requires the designer to develop an ad hoc reprogramming framework. **A well-documented software development framework that can be integrated with commonly used reprogramming tools can facilitate the implementation, evaluation, and use of incremental**

code update algorithms — our secondary focus in this area of this dissertation.

1.1.3 Data Storage

Typical WSN deployments consist of a large number of resource-constrained sensor nodes tasked with monitoring local conditions. Applications such as disaster management, structural health monitoring, and volcanic activity monitoring [15, 34, 67] require constant recording, transmission, and processing of sensed data. Some applications require immediate transmission [15], while others require periodic transmission. The ability to store data in-situ is essential in both scenarios [16]. Applications with immediate transmission requirements may have periods of radio connectivity loss, during which they may need to buffer sensed data. Other applications may employ buffering with periodic transmissions to extend battery life.

Storing buffered data in RAM is not always feasible, as the size of primary memory in typical sensor nodes is often small [20]. For example, the Atmel ATMega644P microcontroller (MCU), used in the MoteStack [22], has only 4KB of RAM [8]. The maximum supported RAM size in a Texas Instruments MSP430 MCU, used in the Telos platform [50], is 64KB [28]. While flash memory is slower than RAM, it offers a viable alternative due to its larger memory size. However, some sensor applications have the capability to acquire data at continuously high rates, overwhelming flash memory, which offers comparatively slow write-erase speeds. Flash memory is also characterized by a limited number of write-erase cycles, potentially "wearing out" due to frequent writes. These limitations have prompted efforts to develop improved storage technologies. Non-Volatile Static RAM (NVSRAM), Ferroelectric RAM (FeRAM/FRAM), Ovonic Unified Memory (OUM), and Magnetoresistive RAM (MRAM) are potential successors of flash technology – all boasting performance advantages, including low power consumption, a large number of write-erase cycles, and fast read and write performance. **Exploiting the performance and feature advantages of these newer technologies is the key to designing fast, lightweight, and reliable file systems — our final focus area**.

1.2 Research Approach and Contributions

In this section, we outline our approaches to addressing the challenges and our contributions.

1.2.1 Task Scheduling

We present the design and implementation of four scheduling systems which support task scheduling in AAS embedded applications. The first is a *basic* scheduler that parallels the design of existing embedded task schedulers. The second is an $O(1)$ scheduler, loosely based on the similarly titled Linux 2.6.8.1 task scheduler. The third is an $O(n)$ scheduler, which improves on the $O(1)$ scheduler by introducing constant time task identification and linear-time rescheduling. Finally, the *intelligent sleep* scheduler uses variable length sleep periods between tasks to achieve an even lower power consumption profile.

We emphasize that these designs are practically motivated. They evolved over the course of 18 months while developing a large-scale environmental monitoring network deployed in the City of Aiken, South Carolina [22]. In 2011, the city's stormwater treatment system was redesigned to reduce the environmental impacts associated with stormwater runoff. The monitoring network was installed in targeted areas throughout the city to monitor the modified treatment system. Our sub-team was responsible for the design of the wireless sensor platforms and the associated firmware used to construct the network. The design process was guided by the need to support continuous, uninterrupted data collection in the face of unattended operation (since Aiken is relatively remote). Maximizing the lifetime of our almost-always-sleeping system was a principal goal. In addition to yielding a successful network deployment, the experience resulted in the first systematic analysis of AAS schedulers, which we present here.

1.2.2 Network Reprogramming

We first present *VSPIN*, a Linux-based framework for developing and testing incremental code update mechanisms to support efficient reprogramming of wireless sensor networks. We developed VSPIN to facilitate the implementation and evaluation of incremental code update algorithms by providing a unified development framework for WSN designers. The current implementation is tailored for reprogramming Atmel AVR MCUs, which have on-chip, in-system reprogrammable flash program memory, using *Avrdude* [17]. However, the solution design can be easily adapted for use with

any MCU which supports in-system programming and uses a set of standardized reprogramming tools.

VSPIN is implemented in Linux, but can be extended to other operating systems. The VSPIN framework consists of a virtual serial port kernel device driver, a user space program, and a boot loader executing on the AVR MCU core. VSPIN connects to the MCU through any communication device which terminates with a serial interface (SPI or UART) on the sensor node. On the host side, it is able to use wireless communication devices (Wi-Fi and ZigBee), or wired communication devices — anything capable of exporting a serial device on the Linux platform (RS-232 or USB).

Next, we present an incremental code update mechanism which transmits an *edit map* encoding the differences between old and new program images. We are able to generate the differences between the two files using a divide-and-conquer dynamic programming approach. Our incremental code update solution does not use block level code comparison. As a result, it is able to locate and send differences at byte-level granularity. Our approach is also independent of any program code structure knowledge, and thus provides a platform and programming language-independent solution.

We adapt Hirschberg's Algorithm [26] (used for computing *maximal common subsequences*) to compute the differences between two program image files. The adapted code differencing algorithm is capable of generating the *diff*s between the two files in $O(n^2)$ time and $O(n)$ space, where n is the length of the new program image. The first step of the incremental code update strategy, i.e. the code differencing algorithm, is run on a standard desktop system to avoid computationally-expensive operations from executing on the sensor nodes.

In the second step, the differences are encoded in an edit map using heuristic-based optimization. The optimization strategy efficiently encodes the edit map using a minimal number of bytes. The edit map is then propagated to the resource-constrained sensor nodes using a standard data dissemination algorithm. The nodes are responsible for decoding the edit map and performing the required data write, and/or move operations to update the program image. Since edit map creation, propagation, and decoding are decoupled, our strategy can be adapted for use with any data dissemination protocol.

1.2.3 Data Storage

Memory technologies that combine the read and write speeds of RAM with the non-volatility of flash can be used to design fast and reliable file systems. In the past, traditional EEPROM technologies have been used to achieve file system reliability in embedded systems [16]. EEPROM differs from flash in that it can be reprogrammed one byte at a time. However, EEPROM is slower than flash, which supports page-based writes and block-based erases; EEPROM is not suitable for use in applications where fast data update rates are required.

Table 1.1: Memory Technology Comparison

	DRAM (MD51V65165E)	SRAM (71V016SA10)	FLASH (SST39LF010)	NVSRAM (CY14B104NA)	OUM[1]	MRAM (MR2A16A)	FRAM (FM22L16)
Non Volatile	No	No	Yes	Yes	Yes	Yes	Yes
Non Destructive Read	No	Partial	Yes	Yes	Yes	Yes	No
Direct Overwrite	Yes	Yes	No	Yes	Yes	Yes	Yes
Write Cycles	∞	∞	10^6	$2 * 10^6$	10^{12}	∞	$>10^{14}$
Read Cycles	∞	∞	∞	∞	∞	∞	$>10^{14}$
Write Speed	$84ns$	$10ns$	$200\mu s$	$20ns$	$10ns$	$35ns$	$110ns$
Read Speed	$84ns$	$10ns$	$45ns$	$20ns$	$20ns$	$35ns$	$110ns$
Erase Speed	$84ns$	$10ns$	$18ms$/sector	$20ns$	$50ns$	$35ns$	$110ns$
Active Current	$120mA$	$65mA$	$5mA$	$65mA$	n/a	$105mA$ (AC)	$8mA$
Stand-by Current	$2mA$	$10mA$	$1\mu A$	$5mA$	n/a	$18mA$ (AC)	$5\mu A$

Table 1.1 presents a comparison of the memory technologies currently available or under development. For a memory technology to be used in a resource-constrained environment, the memory device should have low active and stand-by current consumption. To support a fast file system, the device should provide fast read and write access times. Finally, the memory technology should support a large number of read/write-erase cycles to avoid wearing out. While SRAM and DRAM allow fast read and write access ($84ns$ and $10ns$, respectively), they are volatile in nature, i.e. they lose their data when power is removed. Some older NVSRAM devices achieve non-volatility using battery backed SRAM [19], while newer devices use non-volatile elements (EEPROM cells) [53]. NVSRAM devices typically have data access speeds comparable to SRAM devices ($20ns$). However, NVSRAM allows limited write cycles (like flash) and exhibits high active and stand-by current draw, which makes the technology unsuitable for many sensing applications. In comparison, FRAM [54] offers a high number of read and write cycles (10^{14}), along with fast data access ($110ns$ for reads, writes, and erases). FRAM is characterized by low active and stand-by current consumption ($8mA$ and $5\mu A$, respectively). Unlike flash, FRAM also allows direct byte overwrites.

We present the design of LoggerFS – a fast, lightweight, and reliable file system for embedded network systems. The implementation uses a hybrid memory design consisting of RAM, FRAM, and

[1]Under development

flash. LoggerFS provides fast sensor data access, while imposing only a small main memory footprint. LoggerFS adapts a log-structured file system design, supplemented with a lightweight consistent data structure update mechanism to provide file system reliability in the presence of faults. The design includes a novel FRAM-based caching mechanism which supports flash wear-leveling and fast data buffering during periods of high-rate sensing. Finally, we present a performance evaluation of the LoggerFS implementation on a prototypical in-situ sensing platform.

1.2.4 Contributions

We present the following contributions.

1.2.4.1 Contribution 1 — *Task Scheduling*

We detail the design and implementation of four progressively more efficient scheduling systems designed to support AAS embedded applications; the designs are applicable to virtually any modern MCU. (For the sake of presentation, we focus on the popular *ATmega* family of devices, used in a number of sensor networking platforms [59, 61, 60].) For each scheduler implementation, we present a closed-form algebraic model that captures the scheduling overhead as a function of task load and other parameters. We then use these models to characterize the comparative performance among the designs. To supplement this analysis, we also conduct physical power profiling studies using an ATmega644-based sensor networking platform. The results provide a clear picture of the power consumption profile associated with each design, as well as the comparative lifetime benefits they provide.

1.2.4.2 Contribution 2 — *Network Reprogramming*

We first present the design and implementation of the VSPIN framework. Next, we present the design and implementation of the incremental code update mechanism. We conduct experiments for various representative code update scenarios and present corresponding results to demonstrate the reduction in required reprogramming data and associated energy compared to simple reprogramming strategies involving transmission of full program images. We also compare the results with reductions achieved by other incremental code update strategies described in prior work. We demonstrate significant data and power savings over the state-of-the-art incremental update strategies across a range of code update scenarios.

1.2.4.3 Contribution 3 — *Data Storage*

We present the design and implementation of LoggerFS and its associated APIs. We describe the consistent data structure update mechanism used to achieve data persistence and file system reliability. Next, we present the design of an FRAM-based, write-back cache which allows LoggerFS to efficiently handle bursts of writes. We demonstrate the fault tolerance features of LoggerFS for various representative system and sub-system failure scenarios. We also evaluate the read and write performance of a LoggerFS prototype on an in-situ sensing platform. We demonstrate significant improvements in read and write performance with the FRAM-based, write-back cache, over the LoggerFS implementation without cache.

1.3 Dissertation Organization

The remainder of the dissertation is organized as follows. Chapter 2 presents background material related to task scheduling, network reprogramming, and data storage challenges. Chapter 3 describes the task scheduling contributions. Chapter 4 describes the network reprogramming contributions. Chapter 5 describes the data storage contributions. Chapter 6 discusses elements of related work in the relevant areas. Finally, Chapter 7 concludes with a summary of the contributions and expected impact.

Chapter 2

Background

In this chapter, we present background material on task scheduling, network reprogramming, and reliable data storage in WSNs.

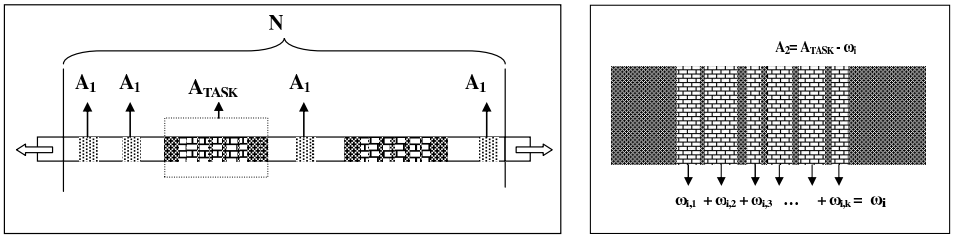
2.1 Task Scheduling

The smallest unit of work that may be scheduled in an AAS system is a *task*, an action taken in response to a timer event. Two specific cases must be considered when characterizing the overhead of any AAS scheduler design. When a scheduler wakes and has no tasks to execute, a small amount of time is expended, referred to as the *null activation period*, denoted by A_1. The amount of time expended when the scheduler wakes and there are tasks to execute, including task execution time, is referred to as the *task activation period*, denoted by A_{TASK}.

In a given time period N, a scheduler experiences A_1 and A_{TASK} multiple times and sleeps the rest of the time. The number of times the scheduler experiences A_1 and A_{TASK} in a time period N is given by n_1 and n_2, respectively. Each instance, i, of A_{TASK} within N consists of time spent executing the task functions, given by ω_i, and the rest of the time expended prior to, in between, and after task execution, denoted by A_2. The relationship between A_1, A_2, A_{TASK}, ω_i, and N is illustrated in Figure 2.1. The total time spent executing task functions in time period N is given by W, calculated as the sum of all ω_i, where $i = \{1, 2, ..., n_2\}$. In the i^{th} occurrence of A_{TASK}, ω_i is calculated as the sum of all $\omega_{i,j}$, where $j = \{1, 2, ..., \mathtt{n_{executed}}\}$; $\mathtt{n_{executed}}$ denotes the number of task functions executed in the i^{th} task activation period. The total time taken to execute all task

functions, W, in time period N is calculated as:

$$W = \sum_{i=1}^{n_2} \sum_{j=1}^{\mathtt{n_{executed}}} \omega_{i,j} \tag{2.1}$$



(a) Components of N (b) A_{TASK} Expanded

Figure 2.1: A_1, A_2, A_{TASK}, ω_i, and N

The *scheduler load* α is the fraction of time the system is either busy scheduling tasks or executing them within time period N. The *task load* β is the fraction of time the system is busy executing just the task functions, given by W, within time period N. α can then be expressed as[1]:

$$\begin{aligned} \alpha &= \frac{(\mathtt{n_1 A_1 + n_2 A_{TASK}})}{\mathtt{N}} \\ &= \frac{(\mathtt{n_1 A_1 + n_2 A_2 + W})}{\mathtt{N}} \\ &= \frac{\mathtt{n_1 A_1 + n_2 A_2}}{\mathtt{N}} + \beta \end{aligned} \tag{2.2}$$

Objective. In an ideal scheduler, with no scheduling overhead, $\alpha = \beta$. To minimize the value of α, both A_1 and A_2 need to be minimized. Our objective is to design a scheduler with the least possible A_1 value; since $n_1 \gg n_2$ in AAS systems, a lower A_1 value, even at the expense of a higher A_2 value, will help in maximizing the efficiency and battery life expectancy of a scheduler.

2.2 Network Reprogramming

Avrdude [17] is a popular Linux-based command-line tool capable of programming flash and EEPROM memory, as well as the fuse and lock bits of an AVR MCU. Avrdude supports specialized hardware programmers which comply with programming protocols specified in AVR068, AVR069,

[1]Assuming n_1, n_2, W, and N are fixed, $n_2 A_{TASK} = n_2 A_2 + W$.

and AVR910 [6, 4, 5], by Atmel, including the popular AVRISP and AVRISPmkII devices. Avrdude also works with a variety of other hardware programmers which connect to the host system using a parallel (i.e., ppi, parport) or serial port.

Listing 2.1 presents three different usage scenarios where Avrdude is used for programming a sensor node. The first example presents a typical scenario using an AVRISP mkII programmer; the subsequent examples show its usage when using VSPIN. Avrdude takes as input the type of MCU being programmed (-p), the communication port (-P), the programmer type (-c), and the input file containing the binary application data (-U) to be written to flash memory (flash:w). The application data (*app.hex*) sent to the boot loader is typically in the 16-bit Intel HEX format [29]. First, Avrdude performs checks to confirm the presence and status of the communication port and the programmer. Next, it queries and checks the values set on the various fuse bits in the MCU. (Fuse bits are stored in specialized, non-volatile registers and control the basic behavior of the MCU.) Finally, Avrdude parses the HEX file and transfers the data.

```
1 // Usage with AVRISPMKII programmer
2 avrdude -p m644 -P usb -c avrispmkII -U flash:w:app.hex
3
4 // Usage with VSPIN boot loader, no kernel module, and no user process
5 avrdude -p m644 -P /dev/ttyUSB0 -c stk500v2 -U flash:w:app.hex
6
7 // Usage with VSPIN boot loader, kernel module, and user process
8 avrdude -p m644 -P /dev/vspins -c stk500v2 -U flash:w:app.hex
```

Listing 2.1: *Avrdude* usage example

Depending on the type of programming mechanism in use, application data can either be transferred to an intermediate device programmer, or sent directly to the MCU to be programmed. If an intermediate programmer is used, the programmer is responsible for transferring the data to the MCU. In the absence of a programmer, the sensor node requires a specialized boot loader capable of communicating with Avrdude and emulating the behavior of a programmer. After the application data has been written to the sensor node, Avrdude compares the MCU fuse values from before and after programming to check for consistency. Finally, it reads the data back from program flash memory, conducts a byte-by-byte comparison with the original binary file, and reports the result of the comparison.

Commercially available hardware programmers require system designers to use a brute force reprogramming approach, i.e., the entire program image must be sent to the target hardware. As a

result, incremental code update mechanisms designed to achieve more efficient network reprogramming rely on ad hoc methods of binary image differencing and data transmission. Specialized boot loaders are used to support the transfer and decoding of the resulting code increments, as well as subsequent reconstruction and programming of the application data image. The boot loader is initially installed on the MCU using a standard programmer. After the boot loader has been installed, it interacts with the host system using a wired/wireless, serial/parallel communication device, using strategy-specific data transmission protocols.

In contrast, VSPIN provides a transparent solution to enable incremental sensor network reprogramming by allowing the use of Avrdude without requiring any changes to its code base or usage (Listing 2.1). VSPIN uses the *STK500 Communication Protocol* [4], supported by Avrdude, to communicate between the host system and the boot loader. As a result, VSPIN is capable of allowing the use of Avrdude in incremental, as well as non-incremental programming modes.

Network reprogramming consists of transferring a program image developed and compiled on a desktop system to networked sensor nodes. ISP-free reprogramming strategies — *wired or wireless* — usually require installing a boot loader on the target device, which receives the application program image. The transferred data is then stored in application flash or external memory [44].

The host development system is responsible for reading the binary image file and extracting the machine code to be injected into the network. The machine code is encoded in the message format specified by the data dissemination protocol used to transfer the data to sensor nodes. The data dissemination protocol is required to be resistant to packet loss and other network issues [51].

Finally, the boot loader executing on each sensor node decodes the messages received and stores the machine code to on-chip flash program memory. Since flash writes occur one page at a time (256 bytes in most embedded devices), the boot loader typically uses a flow control mechanism to control the inflow of data.

While wireless methods of data dissemination eliminate the need for direct connections to the host system, wireless data transmission is energy-intensive. Previous studies have reported that transmission of a single bit of data requires 1,000 times the energy required for the execution of a single instruction[51, 71, 56]. Incremental code update strategies significantly reduce the amount of data that must be transferred to the boot loader, thus improving the energy footprint of the network.

2.3 Data Storage

Designing a file system for sensor systems poses unique challenges due to the scarcity of computational resources afforded by most hardware platforms. The small size of primary memory (RAM) and limited battery power require a reevaluation of common file system design choices, which tend to be resource-intensive.

2.3.1 Batch Data Transmissions

WSNs typically consist of resource-constrained sensor devices that sense and transmit data to one or more base stations, where resource-intensive computations are performed. Since data transmission is energy-intensive, a common strategy is to store data locally (in RAM), and to later send batched data. With the advent of inexpensive flash memory technology, sensor nodes equipped with flash devices are able to store large amounts of sensed data. There are, however, challenges associated with flash storage. The write granularity of flash memory is one page, ranging from `512B` to `8KB`, while the erase granularity is a block, ranging from `64` to `128` pages. The amount of data (in pages) that can be written to flash is limited by the size of RAM (either in the system or on the flash chip), since the data must first be buffered.

2.3.2 Data Sampling Rates

Mission-critical sensor applications can produce continuous data at high rates. This can overwhelm flash memory devices, which write at slower speeds than the incoming data rate due to slow write-erase cycles. The devices also support writes and erases at larger page and block granularities, respectively, compared to the data sample sizes. Flash memory, which is characterized by limited write-erase cycles, can also "wear out" due to frequent data updates during high-rate sensing. Batching data writes in RAM can help with wear-leveling to a limited extent and can sustain short bursts of high-rate sampling. However, for systems with continuously high sampling rates, this is not a feasible solution. Instead, some form of non-volatile memory which is larger in size than the available RAM may be used as cache.

2.3.3 Data Persistence

It is important for sensor devices deployed as part of mission-critical applications to include a robust file system so that data can be accessed reliably, even across device failures (e.g., power disruptions, operating system crashes, etc.) Designing such a file system usually requires check-pointing/snapshotting [57]. However, even the simplest check-pointing/snapshotting solutions are resource-intensive, not suited for implementation on sensor nodes.

2.3.4 Data Considerations

To design a fast file system for sensor nodes, it is important to understand the types of data stored on these nodes, as well as the associated access patterns. WSNs primarily deal with three types of data:

- **Sensor Data:** Data generated by the sensors attached to a node represents the majority of recorded data. In many applications, this data must be batched in RAM or a persistent storage device before it is transmitted to a base station. Some sensors have the ability to generate data very quickly. The sampled data is typically recorded in a sequential manner. During the transmission phase, data is usually read back in a FIFO manner, transmitted, and finally deleted from the storage device.

- **Binary Program Image Data:** The ability to reprogram wireless sensor nodes in large installations is important. Some network reprogramming strategies involve transferring an entire binary program image to the sensor nodes [33, 62, 27]. Some of the more sophisticated, incremental approaches transfer only a *diff* between the new and old images, which is then used to reconstruct the new binary image on each node [45, 31, 32]. Regardless of the reprogramming strategy, the transferred data must be stored in a reliable manner. Errors in binary program image data can cause sensor nodes to stop functioning.

- **Configuration Data:** Configuration data contains the parameter values necessary to configure the operation of sensor nodes (e.g., sampling frequency, file size, transmission rate, etc.) Integrity of this data is important since faulty values can cause sensor nodes to malfunction or stop working altogether. However, configuration data is not updated as often as sensor data. Configuration data is usually stored in EEPROM, in the absence of other persistent storage

devices; reads and writes to EEPROM are usually energy-intensive and slow.

The number of files that must be supported by a file system for sensor nodes is limited, as a substantial percentage of the associated operations are typically for storage and retrieval of sensed data. The number of files required in such a system is on the order of the number of available sensors. Configuration data requires limited additional memory, while the space required to store a copy of a binary program image cannot be greater than the size of the device's programmable flash memory.

Chapter 3

Task Scheduling

In this chapter, we present the design and implementation details, performance models, and evaluation results for four progressively efficient task schedulers designed to support AAS embedded applications.

We focus on a canonical implementation of an AAS scheduler, where a task is composed of a function pointer, a task type, a period, and a due date. The function pointer points to the executable `task` body. The task `type` is either `one_shot` or `periodic`, corresponding to a task that expires after it has been executed, and a task that is continually rescheduled, respectively. The `period` specifies how often the task should be activated. The due date records the time at which the task should next occur.

The basic scheduling functions in our implementation are `scheduler_init`(), `schedule_task`(), and `scheduler_run`(). `scheduler_init`() handles scheduler initialization during system start-up, and `schedule_task`() is used to schedule new tasks. The system spends much of its lifetime in `scheduler_run`(); it contains the core of the scheduling logic and is invoked to start the scheduler.

Our scheduler designs depend on the target hardware system, particularly the timer mechanism. The target microcontroller implements the system clock using an 8-bit counter register, driven by an external `32.768KHz` oscillator. A prescaler of `128` results in an overflow interrupt being triggered once per second; this suspends the executing instruction and begins the interrupt service routine (ISR), where system time is updated. If the processor is in a sleep state, it wakes and enters the ISR. Upon completion, the processor resumes execution following the call to `sleep`().

```
void scheduler_run() {
    while(true) {
        bool task_executed;
        do {
            task_executed = false;
            uint32_t current_time = current_system_time();
            uint8_t task_index;
            for(task_index = 0; task_index < TASK_QUEUE_CAPACITY; task_index++) {
                // if the current (non-empty) task is due
                if((system_task_buffer[task_index].task != NULL) &&
                   (current_time >= system_task_buffer[task_index].due_date)) {
                    // execute the task function
                    (*system_task_buffer[task_index].task)();
                    // handle rescheduling / removal
                    if(system_task_buffer[task_index].type == ONE_SHOT) {
                        system_task_buffer[task_index].task = NULL;
                    } else {
                        system_task_buffer[task_index].due_date +=
                            system_task_buffer[task_index].period;
                    }
                    task_executed = true;
                }
            }
        } while(task_executed);
        set_sleep_mode(SLEEP_MODE_PWR_SAVE);
        sleep_mode();
    }
}
```

Listing 3.1: `scheduler_run()` (Basic Scheduler)

3.1 Basic Scheduler

We present a basic AAS scheduler implementation that parallels the design of existing embedded task schedulers [39, 38, 25]. `system_task_buffer`, an `N`-element array, is initialized with *NULL* entries within `scheduler_init()`. `schedule_task()` finds the first empty slot and stores the task passed as argument.

`scheduler_run()`, shown in Listing 3.1, iterates indefinitely in the outer `while` loop. In each iteration, referred to as an *execution cycle*, the scheduler steps through `system_task_buffer` and executes each task with an expired due date. When a `one_shot` task completes, the task is removed from `system_task_buffer`. When a `periodic` task completes, its due date is updated based on its period. When there are no tasks to execute, the scheduler enters its sleep cycle.

This simple scheduler has a significant power consumption footprint due to the time required to determine whether there are tasks to execute. Even when there are no tasks to execute, the

scheduler wakes and cycles through the entire task buffer. Since the time expended is bounded by N, an increase in task capacity degrades system performance. A scheduler that could perform a constant time lookup into the task array for available tasks would be more desirable.

```
void scheduler_run() {
    while(true) {
        bool task_executed;
        do {
            task_executed = false;
            uint32_t current_time = current_system_time();
            uint8_t task_index = 0;
            while((task_index = 16 - ffs(task_bitmap_active)) < 16) {
                if (current_time >= system_task_buffer[task_index].due_date) {
                    task_executed = run_task(task_index,current_time);
                } else {
                    task_bitmap_active &= ~(1 << (15 ^ task_index));
                    task_bitmap_inactive |= (1 << (15 ^ task_index));
                }
            }
        } while(task_executed);
        task_bitmap_active = task_bitmap_inactive;
        task_bitmap_inactive = 0;
        set_sleep_mode(SLEEP_MODE_PWR_SAVE);
        sleep_mode();
    }
}

static inline bool run_task(uint8_t task_index,uint32_t current_time) {
    // execute the task
    (*system_task_buffer[task_index].task)();
    // handle rescheduling / removal
    if(system_task_buffer[task_index].type == ONE_SHOT) {
        task_bitmap_active &= ~(1 << (15 ^ task_index));
    } else {
        system_task_buffer[task_index].due_date +=
                system_task_buffer[task_index].period;
        if(system_task_buffer[task_index].due_date > current_time) {
            task_bitmap_active &= ~(1 << (15 ^ task_index));
            task_bitmap_inactive |= (1 << (15 ^ task_index));
        }
    }
    return (true);
}
```

Listing 3.2: `scheduler_run()` and `run_task()` ($O(1)$ Scheduler)

3.2 $O(1)$ Scheduler

The $O(1)$ scheduler is based loosely on the Linux 2.6.8.1 scheduler[1]. Adapted to our system, when there are no tasks in the queue, the scheduler performs a constant-time lookup and returns to sleep. This scheduler also uses `system_task_buffer` to store scheduled tasks. Two supporting queues are also introduced; the *active task queue* stores tasks which must be executed in the current execution cycle, and the *idle task queue* stores tasks that have been executed, but which must be re-evaluated the next time the system wakes. To achieve constant-time task lookup, the queues are implemented using bitmaps; a 1 at bit position n indicates a task in the n^{th} element of `system_task_buffer`. At system boot time, `schedule_task`() locates the first free index in the task buffer, and the corresponding locations in the active and idle bitmaps are set and cleared, respectively.

In the execution phase, a call to `ffs`() is performed on the active task bitmap, as shown in Listing 3.2 (line 8). The `ffs`() function, provided by the Atmel AVR C library[7], returns the position of the least significant bit set in a 16-bit word; or 0, if none are set. If a task is identified in the active task queue with a due date greater than the current system time, its index position is cleared in the active task bitmap and set in the idle task bitmap. If the identified task has an expired due date, it is executed by `run_task`(), followed by its removal or rescheduling. Task removal entails removal of the corresponding task bit from the active task bitmap. Task rescheduling involves updating the two bitmaps, as well as the due date of the task in `system_task_buffer`. If the new due date is still expired, the task queues are untouched, ensuring that the task is retriggered during the next call to `ffs`(). If the new due date is later than the current system time, its index position is cleared in the active task bitmap and set in the idle task bitmap. At the end of the execution cycle, when the active task queue is empty, the contents of the idle queue are moved into the active queue to prepare for the next execution cycle.

During the execution cycle, if there are no tasks to execute, the scheduler performs an $O(1)$ lookup into the active task queue and returns to sleep. While $O(1)$ run-time is desirable, a large constant results in increased power consumption. We next consider a design that introduces increased overhead when there are tasks to execute, but very little overhead when there are no tasks to execute — our common case.

3.3 $O(n)$ Scheduler

The $O(n)$ scheduler removes the call to the expensive `ffs`() function; it requires constant time to identify a task to execute, and linear time to reschedule the task post-execution.

Tasks are stored as nodes in a linked list instead of the statically allocated task array. *Slab allocation* is implemented using a static block of memory capable of holding `N` task nodes; `task_free_list`, a pointer to the list of free memory within the static memory block; and `task_queue`, a pointer to the linked list of tasks. Task scheduling involves allocating a node from `task_free_list`, populating the node, and inserting the node in `task_queue` based on due date.

```
1  void scheduler_run() {
2      uint32_t system_sleep_cycle_counter = 0;
3      while(true) {
4          bool task_executed;
5          do {
6              task_executed = false;
7              uint32_t current_time = current_system_time();
8              while((task_queue != NULL) &&
9                    (task_queue->due_date <= current_time)) {
10                 // execute the task
11                 task_node_ptr_t task_ptr = task_queue;
12                 (task_ptr->task)();
13                 task_executed = true;
14                 task_queue = task_queue->next;
15                 // handle rescheduling / removal
16                 if(task_ptr->type == ONE_SHOT) {
17                     free_list_free(&task_free_list, (node_ptr_t) task_ptr);
18                 } else {
19                     task_ptr->due_date += task_ptr->period;
20                     insert_task_in_scheduling_queue(&task_queue, task_ptr);
21                 }
22             }
23             system_sleep_cycle_counter = task_queue->due_date - current_time;
24         } while(task_executed);
25         set_sleep_mode(SLEEP_MODE_PWR_SAVE);
26         while(system_sleep_cycle_counter--) {
27             sleep_mode();
28         }
29     }
30 }
```

Listing 3.3: `scheduler_run`() ($O(n)$ Scheduler)

`scheduler_run`(), shown in Listing 3.3, traverses the list of scheduled tasks and executes those that are due. The removal of `one_shot` tasks is handled by freeing the corresponding task node and returning it to `task_free_list`. Rescheduling of `periodic` tasks is handled by updating the

corresponding task's due date and re-inserting the task at the correct position in the priority queue.

Since tasks are ordered by due date, it is straightforward to determine when the next task needs to be executed, just prior to sleeping. When the system is done executing tasks, the difference between the earliest task due date and the current system time is recorded. When the system wakes, this value is used to control the remaining sleep time; a simple check on this value allows the scheduler to decide if there are any tasks to execute and saves it from having to access the node list. The scheduler therefore experiences shorter wake cycles when there are no tasks to execute.

Since an AAS system typically wakes to find nothing to execute, even a small amount of time expended during a wake cycle can add a measurable performance penalty. With the given hardware and interrupt design, where the processor must wake every second, this is the best performance that could be achieved. However, a scheduler capable of altering the interrupt behavior could yield even better performance.

3.4 Intelligent Sleep Scheduler

The basis of the Intelligent Sleep Scheduler (ISS) is the $O(n)$ scheduler, with updates to the wake, sleep, and clock logic. The central idea is that the rate at which the overflow interrupt is generated can be changed by choosing a different clock prescaler, thus making the duration of the processor sleep period tunable. The clock prescaler can be set to 128, 256, or 1024, so that overflow interrupts are triggered at 1, 2, and 8 second intervals, respectively.

Listing 3.4 presents the `scheduler_run`() implementation. The difference between the earliest task due date and the current system time is recorded at the end of each execution cycle. The system then invokes `intelligent_sleep`(), which partitions this value into multiple divisors, so as to calculate the least number of sleep cycles that can be created from 1, 2, and 8-second intervals. However, changing the rate at which the interrupt is fired is non-trivial.

The current rate at which the interrupt is triggered is called an *epoch*. Changing the clock prescaler (and the epoch) at an arbitrary time causes the 8-bit counter register to contain a value greater than 0, accounting for the partial second of elapsed time since the last overflow interrupt. Since epoch values vary over time in this design, the semantics of this *partial time* also vary. Let the epoch be e_1 at time t_1, when the overflow interrupt is triggered. Let the epoch assume the value e_2 at t_2. Partial time is defined as $(t_2 - t_1)$, calculated as a function of e_1 and the value in the 8-bit

counter register when the epoch was changed to e_2. Partial times for each epoch (i.e. 1, 2, 8) are stored in an array.

```
void scheduler_run() {
    uint32_t system_sleep_cycle_counter = 0;
    while(true) {
        bool task_executed;
        do {
            task_executed = false;
            uint32_t current_time = current_system_time();
            while((task_queue != NULL) &&
                  (task_queue->due_date <= current_time)) {
                //... same as O(n) scheduler ...
            }
            system_sleep_cycle_counter = task_queue->due_date - current_time;
        } while(task_executed);
        intelligent_sleep(system_sleep_cycle_counter);
    }
}

inline void intelligent_sleep(uint32_t int_system_sleep_counter) {
    int_system_sleep_counter = int_system_sleep_counter - 1;
    // determine the number of 1, 2, and 8 second sleep cycles
    // 1 second sleep required?
    sleep_cycle[0] = (int_system_sleep_counter & 0x1);
    // 2 second sleep required?
    int_system_sleep_counter >>= 1;
    sleep_cycle[1] = (int_system_sleep_counter & 0x1);
    int_system_sleep_counter >>= 1;
    sleep_cycle[1] += ((int_system_sleep_counter & 0x1) << 1));
    // 8 second sleep required?
    int_system_sleep_counter >>= 1;
    sleep_cycle[2] = int_system_sleep_counter;

    // compute total number of sleep cycles and begin sleeping
    int_system_sleep_counter = sleep_cycle[0]
           + sleep_cycle[1] + sleep_cycle[2];
    set_sleep_mode(SLEEP_MODE_PWR_SAVE);
    do {
        sleep_mode();
    } while(int_system_sleep_counter--);
}
```

Listing 3.4: `scheduler_run`() and `intelligent_sleep`() (Intelligent Sleep Scheduler)

To obtain the least accumulated partial epoch (required for accurate timer operation), the overflow ISR is identified as the optimal place to change the prescaler. Thus, after an execution cycle, the processor enters a 1-second sleep period, waits for the ISR to be triggered, and then changes the prescaler. Listing 3.5 contains the code for the updated overflow ISR. The overflow ISR ensures that the prescaler is set to the 1-second interval for the mandatory sleep cycle after the 2

and 8-second sleep cycles have been executed.

```
#define PARTIAL_TIME_UPDATE() \
       // update system time based on partial time accumulation \
       system_clock_cycles += system_time_fraction*temp_system_time_epoch; \
       if(system_clock_cycles & ~(0xFF)) { \
           system_time += system_clock_cycles>>8; \
           system_clock_cycles &= 0xFF; \
       }

// timer2 overflow handler
ISR(SIG_OVERFLOW2, ISR_BLOCK) {
    // increment system time by current epoch
    system_time += system_time_epoch;
    if (sleep_cycle[0]) {
        // 1-second sleep required; current epoch 1-second
        sleep_cycle[0] = 0;
    } else if (sleep_cycle[1]) {
        // 2-second sleep required; decrement counter, change prescaler if required
        sleep_cycle[1]--;
        if (system_time_epoch != 2) {
            temp_system_time_epoch = system_time_epoch;
            system_time_epoch = 2;
            TCCR2B = (1 << CS22) | (1 << CS21);
            while(ASSR & 0x1F);
            system_time_fraction = TCNT2;
            TCNT2 = 0x0;
            while(ASSR & 0x1F);
            PARTIAL_TIME_UPDATE();
        }
    } else if (sleep_cycle[2]) {
        // 8-second sleep required; decrement counter, change prescaler if required
        sleep_cycle[2]--;
        if (system_time_epoch != 8) {
            //... analogous to above case ...
        }
    } else if (system_time_epoch != 1) {
        // all counters are 0; prescaler reset for mandatory 1-second sleep
        //... analogous to above case ...
    }
}
```

Listing 3.5: Overflow ISR (Intelligent Sleep Scheduler)

At the start of the ISR, the system time is updated using the value of the current epoch. Next, the change of prescaler (and epoch) is performed, if needed. If the clock prescaler is updated, partial time is recorded, and accumulated partial time is calculated as the sum of its previous value and the product of the current partial time and the last epoch value. Since every 256 fractions represents 1 second of time, if accumulated partial time is greater than or equal to 255, the system time is incremented and the accumulated partial time is appropriately updated.

3.5 Algebraic Models

The schedulers were implemented for the MoteStack, a state-of-the-art in-situ sensing platform that uses an ATMega644, 8-bit microcontroller (MCU) operating at 10 MHz. A line-by-line code analysis was performed with the assistance of *AVR Studio*, a cycle accurate device emulator, to derive the closed-form algebraic models.

3.5.1 Basic Scheduler

In the basic scheduler, the null activation period (A_1) is given in μs as:

$$A_1 = 8.9 + 1.5 * \mathtt{n}_{\mathtt{queue\_capacity}} + 1.3 * \mathtt{n}_{\mathtt{in\_queue}} \tag{3.1}$$

where $\mathtt{n}_{\mathtt{queue\_capacity}}$ denotes the capacity of the task queue, and $\mathtt{n}_{\mathtt{in\_queue}}$ denotes the number of tasks in the queue.

A_2 (in μs) is given by the following formula:

$$\begin{aligned} A_2 = {} & 8.9 + 3.1 * \mathtt{n}_{\mathtt{executed}} + 2.6 * \mathtt{n}_{\mathtt{iter}} \\ & + (1.5 * \mathtt{n}_{\mathtt{queue\_capacity}} + 1.3 * \mathtt{n}_{\mathtt{in\_queue}}) * \mathtt{n}_{\mathtt{iter}} \end{aligned} \tag{3.2}$$

Recall that $\mathtt{n}_{\mathtt{executed}}$ denotes the number of task functions executed in the current task activation period; $\mathtt{n}_{\mathtt{iter}}$ denotes the number of times the main scheduler loop executes (Listing 3.1, *lines 4-24*). Assuming that $\forall i,\ (\omega_i + A_2) \leq 1$ *second*, the value of $\mathtt{n}_{\mathtt{iter}}$ is calculated as follows:

$$\mathtt{n}_{\mathtt{iter}} = 1 + \lceil \frac{1}{\mathtt{task\_period}_{\mathtt{min}}} \rceil \tag{3.3}$$

where $\mathtt{task\_period}_{\mathtt{min}}$ is the smallest period value present in the task queue associated with a task that has a due date earlier than the current system time.

3.5.2 The $O(1)$ Scheduler

In the $O(1)$ scheduler, A_1 is given by:

$$A_1 = 14.5 + 24 * n_{in\_queue} + (2.8 * (\lceil \frac{n_{queue\_capacity}}{16} \rceil - 1)) * n_{in\_queue} \quad (3.4)$$

A_2 is given as follows:

$$A_2 = 19.9 + 6.5 * n_{executed} + 24 * n_{in\_queue} * (n_{iter} - 1) + 2.8 * (\lceil \frac{n_{queue\_capacity}}{16} \rceil - 1) * n_{in\_queue}) * (n_{iter} - 1) \quad (3.5)$$

3.5.3 The $O(n)$ Scheduler

The $O(n)$ scheduler has a constant null activation period of 7 μs (A_1). A_2 is given by the following formula:

$$A_2 = 14.4 + (13.7 + t_{ins}) * n_{executed} + 5.6 * (n_{iter} - 1) \quad (3.6)$$

where t_{ins} denotes the time spent within the insertion sort during rescheduling, post task execution.

The value of t_{ins} is given by the following formula:

$$t_{ins} = \begin{cases} 0.2, & \text{if } n_{in\_queue} = 0; \\ 3.7 * [1, n_{in\_queue}) & \text{if } n_{in\_queue} > 0; \end{cases} \quad (3.7)$$

where $[1, n_{in\_queue})$ denotes any value between 1 and (n_{in_queue} - 1).

3.5.4 The Intelligent Sleep Scheduler

The null activation period (A_1) for the ISS is the hardest to analyze due to its complex ISR control flow paths. A flow chart indicating the different paths is shown in Figure 3.1. The value A_1 assumes in a given null activation period depends on the values of the various system variables in that specific period and is given by:

$$A_1 = 0.6 + t_{ISR} \quad (3.8)$$

where $\mathtt{t_{ISR}}$ denotes the amount of time elapsed between the start of the overflow ISR (*line 10*, Listing 3.5) and the start of the scheduling loop in `scheduler_run`() (*line 6*, Listing 3.4).

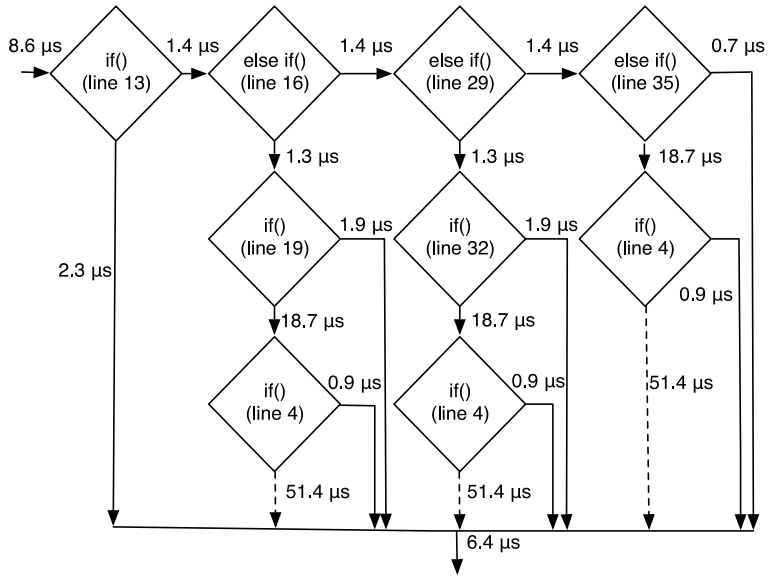


Figure 3.1: ISR Execution Profile ($\mathtt{t_{ISR}}$) (Intelligent Sleep Scheduler)

The accumulation of partial time fractions in the clock update logic requires 51.4 μs. However, this value is ignored for modeling purposes. The latest possible invocation of the partial update logic (Listing 3.5, lines 35-38) is approximately 31.5 μs after the start of the ISR. Thus, the maximum partial time accumulated is approximately 31.5 μs, close to a single oscillation of the external oscillator. Even in the 1-second interval case, the prescaler is set to 128, and the probability of partial time accumulation is small. Even if it does accumulate, for these 31.5 μs intervals to total 1 second, approximately 31,746 occurences of A_1 or A_2 are required. Hence, the time is assumed to be negligible.

A_2 for the ISS is given by:

$$\begin{aligned} A_2 = {} & \mathtt{t_{ISR}} + 13.4 + 5.6 * (\mathtt{n_{iter}} - 1) \\ & + (13.7 + \mathtt{t_{ins}}) * \mathtt{n_{executed}} \end{aligned} \tag{3.9}$$

where $\mathtt{n_{iter}}$, $\mathtt{n_{executed}}$, $\mathtt{t_{ins}}$, and $\mathtt{t_{ISR}}$ are defined as before.

3.6 Evaluation

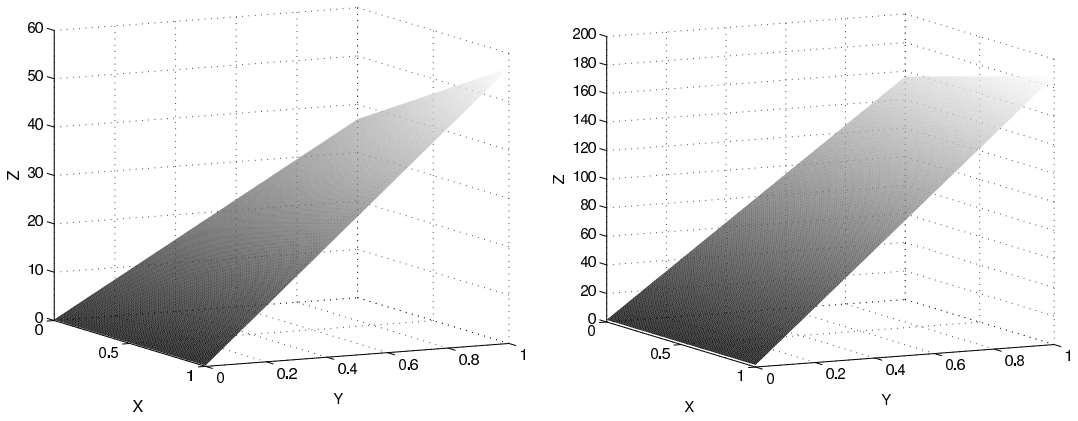
We first consider the performance of the schedulers based on the algebraic models of their behavior. We then measure the scheduler power consumption for a given set of tasks on physical hardware.

3.6.1 Comparative Analysis

We compare the scheduling overhead of each scheduler under varying load conditions; results are shown in Figures 3.2 and 3.3. Due to the number of variables in the equations for A_1 and A_2, we make some assumptions to limit the evaluation space. We fix both $\texttt{n}_{\texttt{queue\_capacity}}$ and $\texttt{n}_{\texttt{in\_queue}}$ to 128, and $\texttt{n}_{\texttt{iter}}$ to 2 (limiting $\texttt{task\_period}_{\texttt{min}}$ to greater than or equal to 1 second – Eq. (5)). We generate the values of $\texttt{t}_{\texttt{ins}}$ using a pseudo-random number generator and fix the values for all subsequent calculations across the schedulers. For each scheduler, we measure the *scheduling overhead*, given by $n_1A_1 + n_2A_2$, in seconds, on the *Z-axis*, when N is set to 500 seconds. N is composed of $(n_1 + n_2)$ 1-second counts. We plot the *fraction of tasks executed* on the *X-axis*, given by $\texttt{n}_{\texttt{task\_executed}}$ over $\texttt{n}_{\texttt{in\_queue}}$, and the *load factor* (given by n_2 over $(n_1 + n_2)$) on the *Y-axis*. The system load factor is helpful in understanding the interplay between A_1 and A_2.

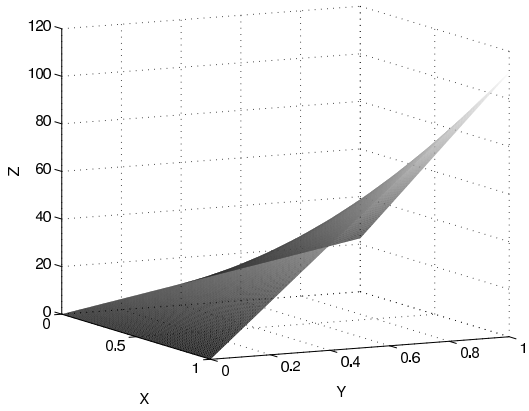
Figures 3.2(a) and 3.2(b) show the results for the basic and $O(1)$ schedulers, respectively. The planar slopes for both graphs are similar, owing to the fact that both schedulers yield A_2 values that depend primarily on similar $\texttt{n}_{\texttt{queue\_capacity}}$ and $\texttt{n}_{\texttt{in\_queue}}$ coefficients. At higher load factors, where $n_2 >> n_1$, the $O(1)$ scheduler performs worse than the basic scheduler, but at lower load factors, the differences are negligible. Figures 3.2(c) and 3.2(d) show the results for the $O(n)$ scheduler and ISS, respectively; again the curves are similar. The $O(n)$ scheduler and ISS incur less overhead than the basic and $O(1)$ schedulers at load factors below 0.8, as they are not dependent on $\texttt{n}_{\texttt{queue\_capacity}}$. We also observe that at higher load factors, the value of $\texttt{n}_{\texttt{task\_executed}}$ affects all schedulers significantly. At lower load factors, both the $O(n)$ and intelligent schedulers exhibit very low overhead (<2% for load factors of 0.3). To further differentiate the two schedulers, we consider their performance at very low load factors, on the order of 0.001, typical in AAS systems.

Since the overhead contribution of A_1 is significantly larger than A_2 at very low load factors, we focus on the impact of A_1 in isolation. In Figures 3.3(a) and 3.3(b), we measure, for each scheduler, the contribution of A_1, given by n_1A_1, on the *Y-axis*, against the load factor, given by

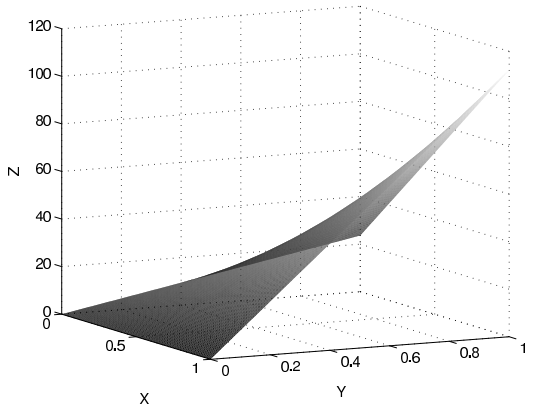


(a) Basic Scheduler

(b) $O(1)$ Scheduler



(c) $O(n)$ Scheduler



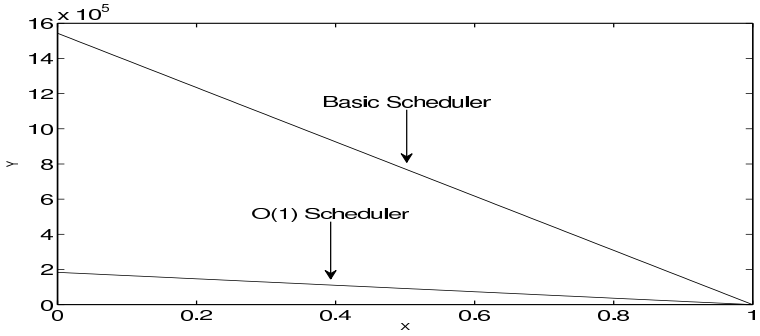
(d) Intelligent Sleep Scheduler

Figure 3.2: X = $\frac{n_2}{n_1+n_2}$, Y = $\frac{\mathtt{n_{task\_executed}}}{\mathtt{n_{in\_queue}}}$, Z = $n_1A_1 + n_2A_2$

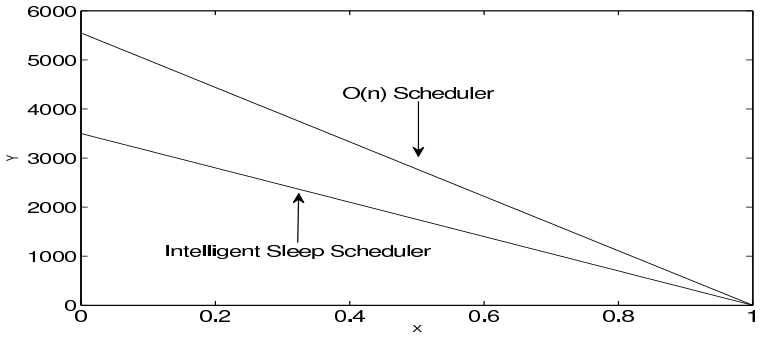
n_2 over $n_1 + n_2$, on the X-axis. With a side-by-side comparison, we see that the basic and $O(1)$ schedulers have a much higher null activation period contribution than the other two schedulers — approximately three orders of magnitude larger. These schedulers are relatively inefficient at lower load factors. We also observe that the ISS performs the best among all the schedulers presented. The explanation is simple: Its ability to sleep for longer periods of time gives the ISS a comparative advantage over schedulers which need to wake every second.

3.6.2 Power Consumption Profile

We now characterize the power consumption profiles of the four schedulers. For this purpose, we installed a test application on the MoteStack device, using each scheduler. The application



(a) Basic and $O(1)$ Scheduler

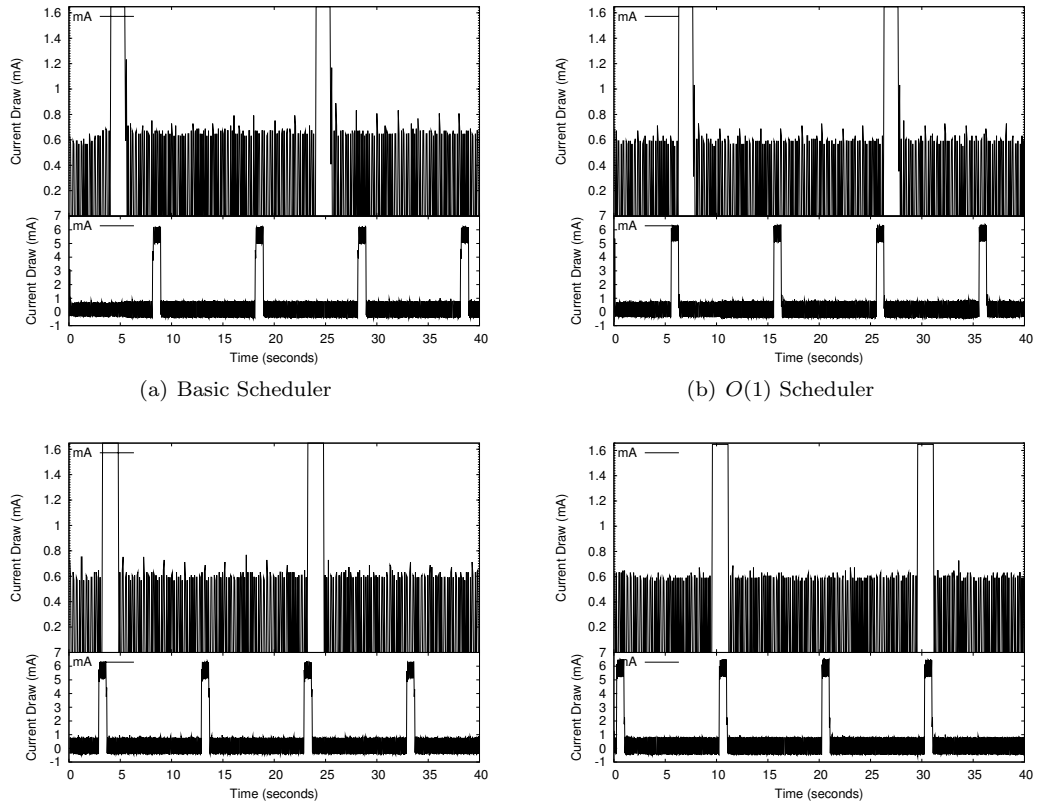


(b) $O(n)$ and Intelligent Sleep Scheduler

Figure 3.3: *Null Activation Period* Contributions (X = $\frac{n_2}{n_1+n_2}$, Y = $n_1 A_1$ μs)

schedules a periodic *null* task with a duration of $750ms$, executed every $10s$. We connected a 10Ω sense resistor in series with the power supply of the MoteStack and measured the voltage difference across the resistor using an oscilloscope. The voltage change is directly proportional to the current draw (and power consumption, when voltage is constant) by Ohm's Law. Figures 3.4(a) – 3.4(d) summarize the consumption profiles for the four schedulers. In each graph, the horizontal axis represents time, and the vertical axis represents current draw. The bottom halves of the figures show the complete consumption profile; the task activation periods are visible. The top halves show a magnified view of the profile, such that the null activation periods can be seen.

We sample data over a 10-second window, which captures current draw values for a single task activation period, multiple null activation periods, and the associated sleep periods. We calculate the average overall and A_{TASK} current draw – the A_{TASK} values vary by scheduler design. The average current draw for the basic scheduler (Figure 3.4(a)) over the window is 0.613 mA (average A_{TASK} current draw is 5.52 mA), while the average current draw for the $O(1)$ scheduler (Figure 3.4(b)) is 0.605 mA (average A_{TASK} current draw is 5.28 mA). The average current consumption



(a) Basic Scheduler

(b) $O(1)$ Scheduler

(c) $O(n)$ Scheduler

(d) Intelligent Sleep Scheduler

Figure 3.4: Scheduler Power Consumption Profiles

for the $O(n)$ (Figure 3.4(c)) and the Intelligent Sleep (Figure 3.4(d)) schedulers is 0.616 mA (average A_{TASK} current draw is 5.56 mA) and 0.603 mA (average A_{TASK} contribution is 5.49 mA), respectively.

Figure 3.5 presents the life expectancy of a $1000mAh$ battery when used to supply power to a MoteStack running the four schedulers under different almost-always-sleeping scenarios. Data for Figure 3.5 was obtained by extrapolating the average current draw and average A_{TASK} current draw from Figures 3.4(a) – 3.4(d) and applying them to applications which sleep for 5, 10, 15, 30 45, and 60 minutes between task executions. We observe that the Intelligent Sleep Scheduler consistently yields higher battery longevity for all applications.

Consider the application which sleeps for 15 minutes between tasks. A MoteStack running this application and drawing its power from a $1000mAh$ battery would last approximately 5,375 hours using the basic scheduler. The same MoteStack would last for 5,380 hours using the $O(1)$

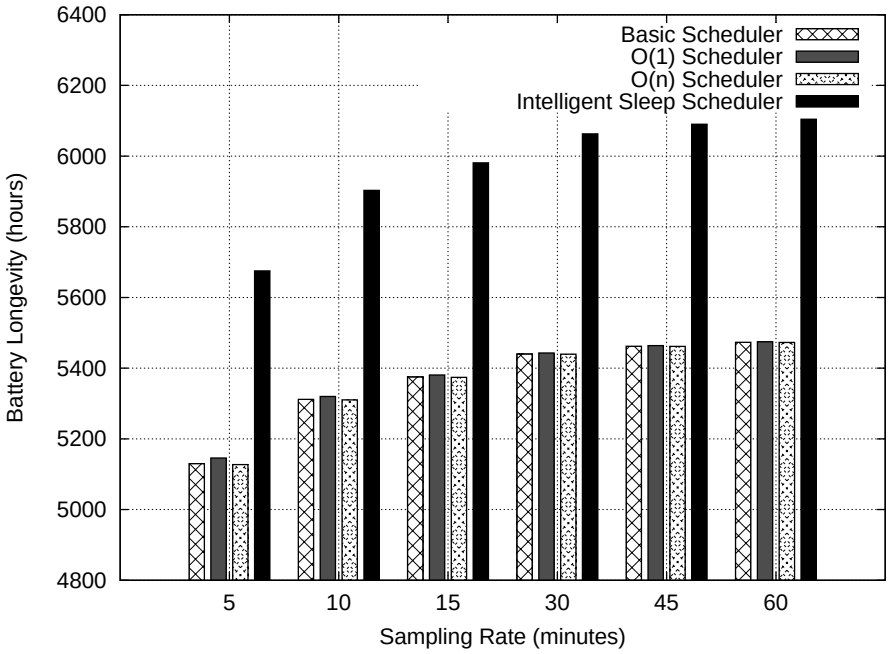


Figure 3.5: Battery Life Expectancy

scheduler. A MoteStack using the $O(n)$ scheduler would last for 5,374 hours, while the ISS offers the longest runtime, of approximately 5,980 hours – 10% longer than any of the other schedulers. Though all the scheduler designs dictate a linear decrease in power consumption with an increase in the time period between task activation periods, not surprisingly, the rate of the decrease for the ISS is higher compared to the others, due to its ability to sleep for longer periods, thus enabling a longer battery life.

3.7 Summary

In this chapter, we presented the design, implementation, and analysis of four progressively efficient schedulers designed to support *almost-always-sleeping* embedded applications. We presented a *basic scheduler*, which uses a rudimentary array to store tasks. We next presented the $O(1)$ *scheduler* based on the Linux 2.6.8.1 scheduler. This design incurs performance penalties due to

an expensive call to `ffs`(). Next, we presented the $O(n)$ *scheduler*, which uses a priority queue to store tasks and improve its tracking of sleep cycles, performing significantly better than the previous schedulers. Finally, we presented the *Intelligent Sleep Scheduler*, which makes use of hardware features to extend physical sleep cycles, further reducing scheduling overhead. We analyzed the runtime of each scheduler and presented detailed performance results under varying load conditions. We found that below a certain load factor, the $O(n)$ and Intelligent Sleep Schedulers work well. However, under lower load factors, the Intelligent Sleep Scheduler performs markedly better than all other designs. This is the first systematic consideration of this increasingly relevant class of schedulers.

Chapter 4

Network Reprogramming

In this chapter, we first present the design and implementation of VSPIN, the Linux-based framework for developing and testing incremental code update mechanisms. We then detail the design and implementation of the incremental code update mechanism developed using VSPIN and present experimental results for various code update scenarios to demonstrate the achieved reduction in reprogramming time and energy consumption. Finally, we compare the performance results with other incremental code update strategies described in prior work.

4.1 VSPIN Framework

VSPIN is a Linux-based framework consisting of a kernel module and a user process. The kernel module and user process execute between Avrdude and the physical serial device, while the boot loader is installed on the sensor node. Figure 4.1 presents an overview of the VSPIN architecture. Using the kernel module and user process, VSPIN has the ability to intercept messages sent by and sent to Avrdude. This is achieved by providing a *virtual serial communication port*, used as the communication port input parameter to Avrdude. The port is emulated by the VSPIN kernel module.

The messages intercepted by the kernel module are transferred to the VSPIN user process, which recognizes the syntax and semantics of the messages specified in the STK500 communication protocol. Depending on the type of message, the user process either forwards the message to the boot loader via the physical communication layer (represented by the `/dev/ttyUSB0` and `ftdi_sio`

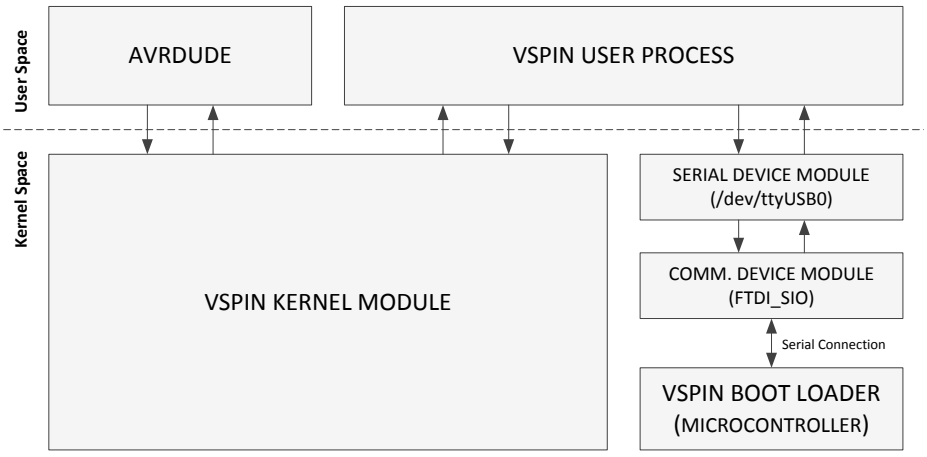


Figure 4.1: VSPIN Architecture

modules in Figure 4.1), or responds to the message itself. If the message is forwarded, the boot loader processes the message and sends the corresponding response back over the serial connection, as per the STK500 protocol. The user process reads the responses sent by the boot loader and relays them back to Avrdude via the kernel module. In addition to its ability to parse and forward messages to the MCU, and to respond to selected messages on its own, the VSPIN user process contains the incremental code update logic under test/development. The update logic resides in a separate module in the user process; it accepts pointers to the old and new program image versions as input, and outputs an edit script. The user process also transmits the information in this edit script to the MCU, where it is used to reconstruct the new program image from the old version. As a result, we are able to "plug-in" any algorithm in VSPIN for incremental sensor network reprogramming.

4.1.1 Kernel Module

The VSPIN kernel module creates two devices on initialization, a virtual serial device, `/dev/vspins`, and a character device, `/dev/vspinc`; each has its own set of file operations. Figure 4.2 provides a detailed overview of the VSPIN kernel module. The virtual serial port (`/dev/vspins`) is used to communicate with Avrdude, and the character device (`/dev/vspinc`) is used to communicate with the VSPIN user process. The devices act as two end points of a communication channel and are used to efficiently channelize the transfer of data from Avrdude to the user process and vice-versa.

VSPIN makes use of two separate devices to solve simultaneous read and write synchro-

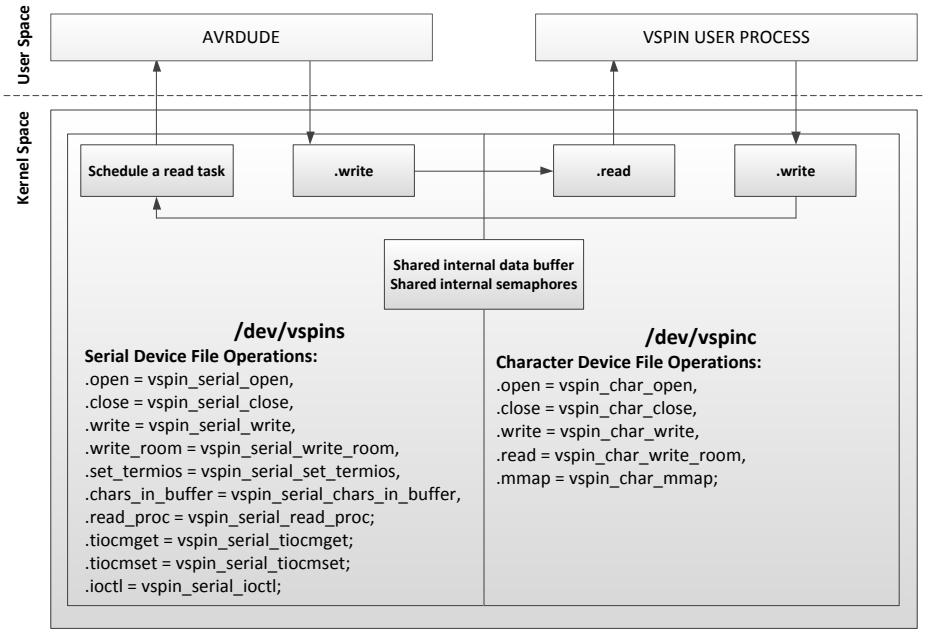


Figure 4.2: VSPIN Kernel Module

nization issues arising from the use of a single device. The `/dev/vspins` device is required because Avrdude expects to communicate with a serial device and issues Linux *termios* library calls to the communication port it connects to. The virtual serial driver implements all of the file operations required to emulate a serial device, as shown in Figure 4.2. It supports all the termios library calls (`cfsetispeed()`, `cfsetospeed()`, `tcgetattr()`, `tcsetattr()`, etc.), as well as `fcntl()` and `select()` calls made from Avrdude. It also provides support for a subset of the options within the `ioctl()` system call.

While the serial device driver is dedicated for use with Avrdude, the character device driver provides a set of file operation callbacks for handling data exchange with the VSPIN user process using `read()`, `write()`, and `mmap()` calls. If VSPIN had been designed with a single virtual device, with simultaneous opens from Avrdude and the user process, channelization of data between Avrdude and the user process would have been substantially more complex. Further, serial device drivers do not provide a `read()` callback [14]; the read mechanism in serial tty drivers operates by forwarding data received from the communication channel to the *flip buffer* in the Linux *tty core* module. Figure 4.3 illustrates the interaction between the Linux tty modules and the VSPIN kernel modules. The tty core module handles data forwarding to the process which has an open on the serial device; the

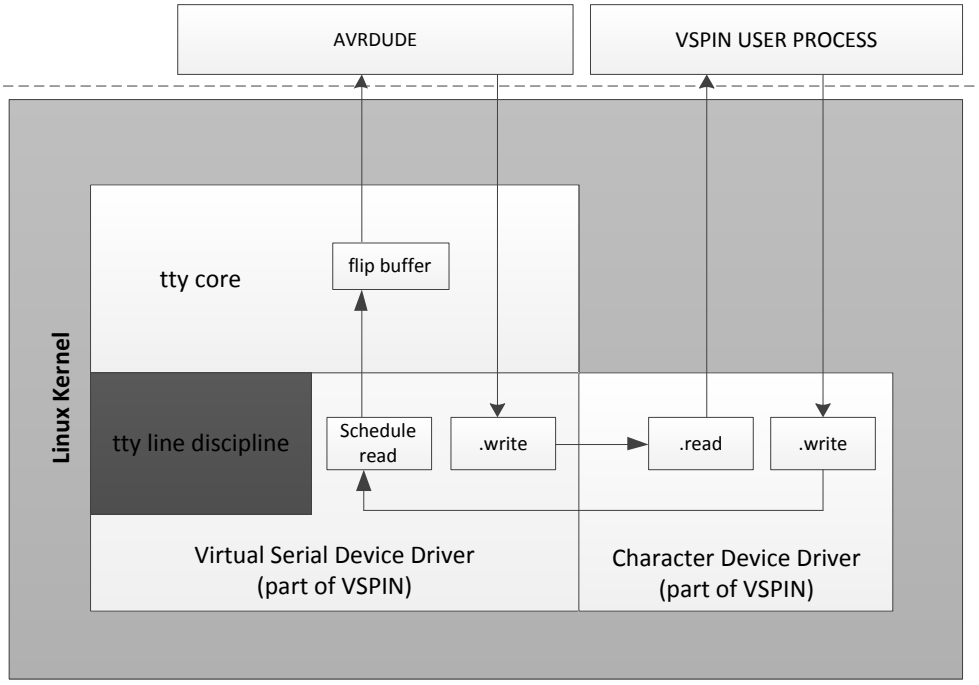


Figure 4.3: Linux *tty* Modules and VSPIN Kernel Module Interaction[1]

behavior is undefined in the event of multiple opens on a single device.

On module initialization, the kernel module handles the registration of the serial device driver with the tty core module and creates all the necessary *sysfs* entries and devices. The serial device driver also interacts with the Linux *tty line discipline* module (Figure 4.3), required for supporting the `select()` call. The kernel module provides a single fixed size buffer which is shared by the two devices and used to transfer data from Avrdude to the user process and vice-versa. The module also provides a semaphore to protect the buffer.

Avrdude communicates with the boot loader by issuing an open on its communication port, `/dev/vspins`, and issuing read and write calls to it. When Avrdude intends to send messages to the boot loader, it writes the message to `/dev/vspins`, the emulated serial port. Immediately after the write, Avrdude issues a timed read on `/dev/vspins` to collect the response to the message sent. When the virtual serial device receives data from Avrdude via the write callback, it stores the data in the shared buffer and hands the buffer over to the read callback of the character device. Similarly, when the character device receives data from the user process via the write callback, it stores the

[1]Reproduced and adapted from [14].

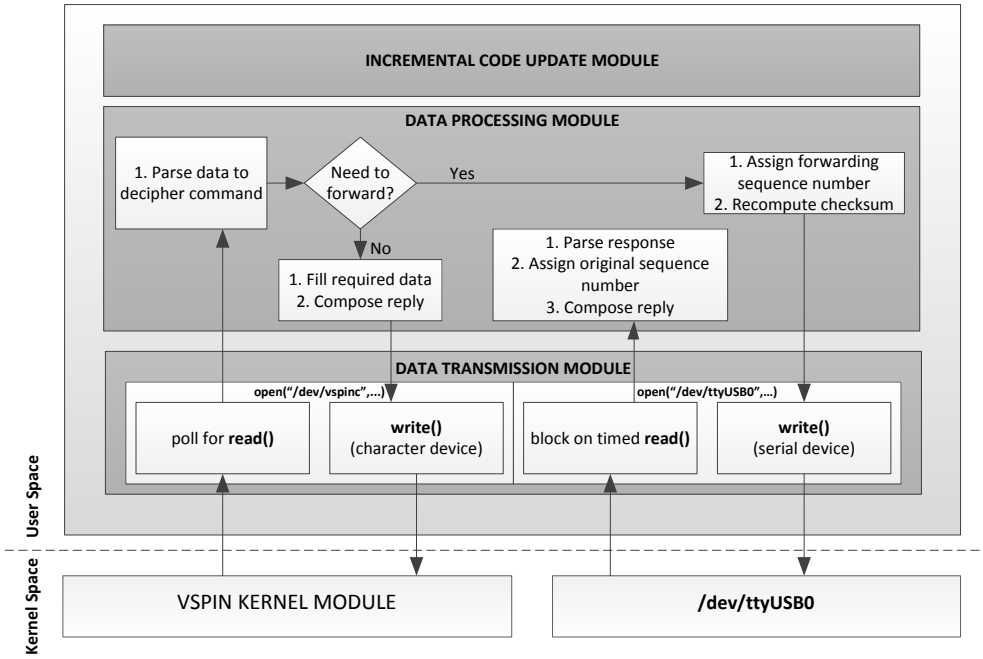


Figure 4.4: VSPIN User Process

data in the shared buffer and hands the buffer over to the virtual serial device. The virtual serial device treats the character device as the communication channel. On receiving data, the serial device schedules a *read task*, which forwards the received data to the flip buffer of the tty core module. The data from the flip buffer is then forwarded to Avrdude, which has been blocking on the timed read call.

4.1.2 User Process

The VSPIN user process consists of three parts: the *transmission module*, the *processing module*, and the *incremental code update module*. Figure 4.4 provides an overview of the user process. The transmission module is responsible for all transmissions to and from the user process. The module issues two opens, one on the `/dev/vspinc` character device to interact with Avrdude via the kernel module, and the other on the `/dev/ttyUSB0` serial communication device to interact with the boot loader.

The transmission module continuously issues polling reads on `/dev/vspinc` for new data. When a message is received, the transmission module transfers the message to the processing module, which deciphers the message sent by Avrdude. Some messages sent by Avrdude inquire about values

which are hard-coded in the boot loader software. In such scenarios, the processing module itself composes a response and sends it to the kernel module using a write call in the transmission module, instead of forwarding the message to the boot loader, thus enabling high speed responses. Messages sent by Avrdude which require a response from the boot loader are forwarded by the processing module to the transmission module, which sends the data through the serial communication channel. Next, the transmission module issues a timed read call on the serial communication channel to read back the response sent by the boot loader. On receiving the response, the transmission module forwards the data back to the processing module, where the response is parsed and any necessary changes are made (as explained below). Finally, the processing module forwards the data to the kernel module through the transmission module.

The STK500 communication protocol uses sequence numbers and checksums for data transmission. Since the user process forwards some messages to the boot loader and responds to other messages directly, it maintains two sets of sequence numbers. One set of sequence numbers is used with Avrdude, while the other is used for communicating with the boot loader. The processing module is responsible for translating the sequence numbers and also adjusting the checksum values caused by sequence number changes.

4.1.3 Incremental Code Update

VSPIN assumes that incremental code update strategies require comparison of the new version of the program image with the old version already installed on the sensor node, and that the old image version is not available on the host system. Figure 4.5 summarizes the incremental code update process used by VSPIN. The process begins when Avrdude starts sending the new program image to the boot loader via the virtual serial device of the kernel module. The data for the new program image is intercepted by the user process and stored in a data buffer d_1. The user process emulates the boot loader behavior and sends a response back to Avrdude confirming receipt of the data. On receiving the confirmation, Avrdude attempts to read the programmed data back from the boot loader. The user process receives the read request and forwards the request to the boot loader. The boot loader reads data from flash memory and sends it back to the user process. The user process receives the old program image data from the boot loader and stores it in another data buffer d_2.

Next, the user process sends data from buffer d_1 to Avrdude instead of from buffer d_2.

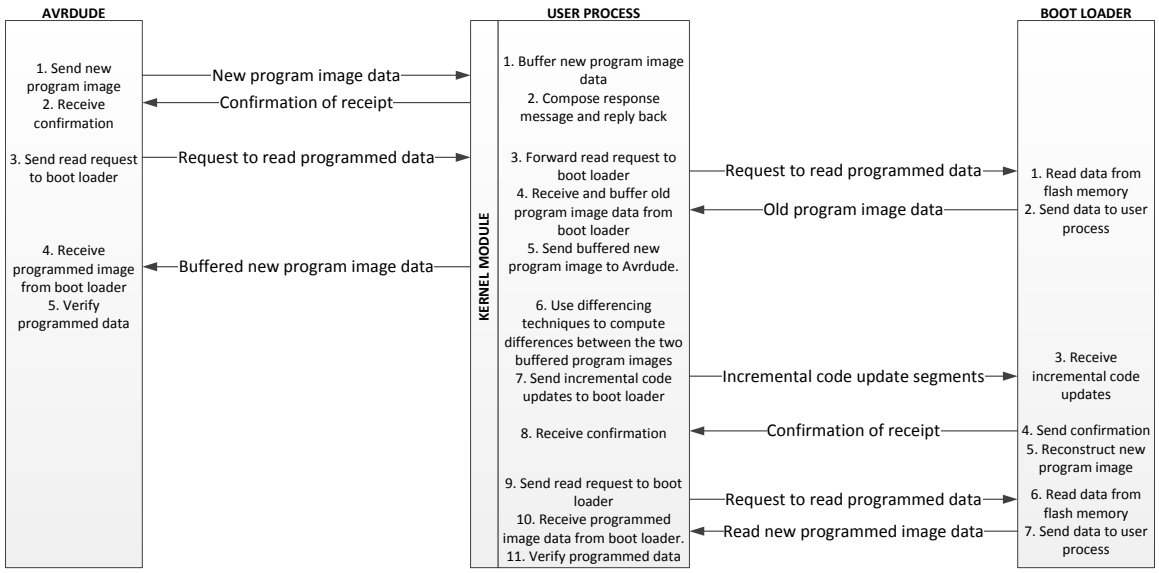


Figure 4.5: Incremental Code Update Process

Avrdude receives the program image, verifies that the program image received is identical to the program image it sent, and exits. At this point, the user process has access to both the old and new versions of the program image in data buffers d_1 and d_2. The two program images are now sent to the incremental code update module inside the user process, where differencing techniques are used to compute the differences between the two binary images. The incremental code update module creates an edit script which encodes the differences and can be used by the boot loader to translate the old program image into the new image. The edit script is sent by the user process to the boot loader. On receipt of the edit script, the boot loader sends a confirmation back to the user process.

Next, the boot loader reconstructs the new program image from the old image in flash memory using the edit script it received. In the final step, the user process attempts to read the (now) programmed data from the boot loader. The boot loader reads the newly reconstructed program image data from flash and sends it back to the user process. The user process uses this data to verify successful reprogramming of the sensor node, failing which it restarts from step 7 in Figure 4.5.

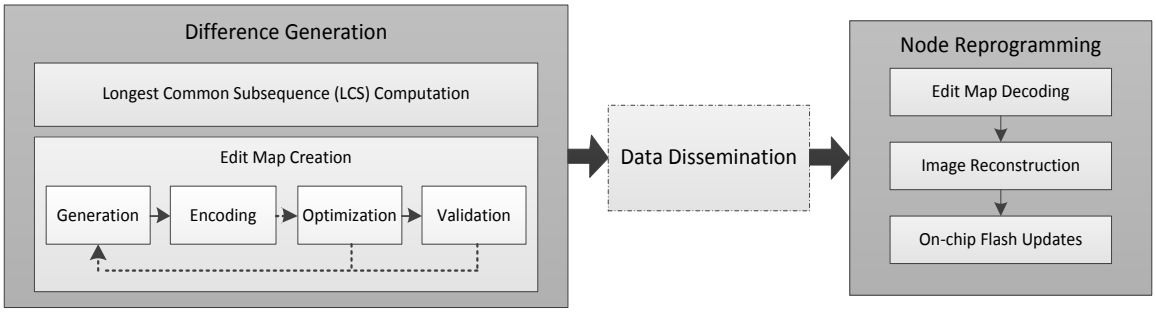


Figure 4.6: Incremental Network Reprogramming Workflow

4.1.4 Boot Loader

The boot loader provided by VSPIN has a simplistic design, closely resembling the boot loader design described in [44]. The boot loader receives messages via a serial interface, interprets the message, reacts based on the command contained in the message, and finally sends a response back via the serial interface. The boot loader supports the set of commands listed in the STK500 communication protocol. In addition to these commands, it supports user-defined commands for incremental code updates, which typically involve receiving the edit script, reconstructing the program image, and finally programming flash.

4.2 Incremental Code Update Mechanism

The incremental code update strategy presented here begins with the idea that an application program image can be thought of as a byte string of length n. When the program image for a sensor node needs to be updated, the *maximal common subsequence*, also known as the *longest common subsequence* (LCS), between the two strings is computed. The substrings of the new program image which are not part of the LCS constitute the image data that must be transmitted to the sensor node.

Figure 4.6 illustrates the workflow for the reprogramming strategy. The *difference generation* phase, which involves computation of the LCS and the associated edit map, is computed at the host system, as it is the most resource-intensive phase of the process. The *data dissemination* phase is initiated by the host system using a standard data dissemination protocol, such as XNP or Deluge, over a wireless radio link. We do not explore the data dissemination algorithms as part of our

approach; these are beyond the scope of our work. The final *node reprogramming* phase is the responsibility of the sensor network.

4.2.1 Difference Generation

The difference generation phase consists of two sub-phases. The first involves computing the LCS between the two program image versions. The second sub-phase uses the LCS to prepare and encode the *edit map* containing the edit script to be transmitted to the sensor network.

4.2.1.1 LCS Computation

The LCS computation is the first sub-phase of difference generation. Consider an arbitrary node in a sensor network. Let the version of the program image currently installed on the sensor node be defined as $X = x_1x_2x_3...x_m$, where $|X|$ is m, and x_i is a byte at offset i in the image. Let the new program version be n bytes long and be defined as $Y = y_1y_2y_3...y_n$. Hirschberg's algorithm [26] finds the string $L = l_1l_2l_3...l_r$, such that L is a common subsequence of X and Y, and its length r ($|L|$) is maximized. Let the set of prefixes of the strings X and Y be $\{X_1, X_2, X_3, ..., X_m\}$ and $\{Y_1, Y_2, Y_3, ..., Y_n\}$, respectively, where X_i and Y_i are the prefixes of size i bytes.

Let $LCS(X_i, Y_j)$ denote the LCS for the prefixes X_i and Y_j. If we denote $|LCS(X_i, Y_j)|$ as $C(i, j)$, the dynamic programming formulation for $C(i, j)$ is as follows:

$$C(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C(i-1, j-1) + 1 & \text{if } x_i = y_i \\ max(C(i-1, j), C(i, j-1)) & \text{if } x_i \neq y_j \end{cases}$$

Figure 4.7 presents a logical view of the LCS L; the program image strings X and Y are represented as m and n byte memory blocks, respectively. L is represented by the sequence of shaded boxes labeled $l_1, l_2, l_3, ..., l_r$. The arrows illustrate the mapping from the old to the new program image segments in the LCS.

Hirschberg initially presents an algorithm to calculate the length of the LCS of any two strings ($C(i, j)$) using dynamic programming, along with a memoization-based, bottom-up, table-building scheme, requiring $O(mn)$ time and $O(mn)$ space [26]. Next, a modified version of this algorithm, capable of computing the LCS length in $O(min(m, n))$ space is presented. Using the

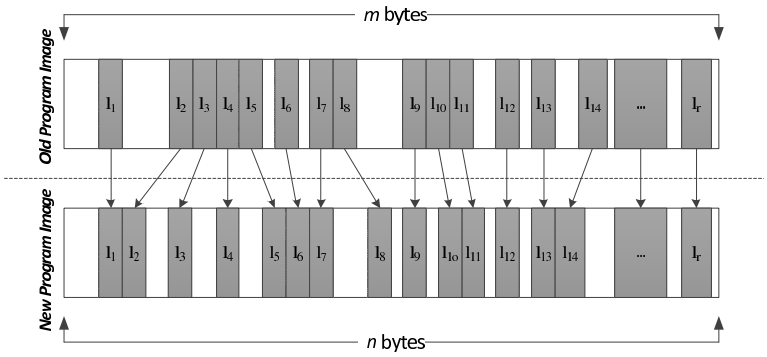


Figure 4.7: Logical View of the LCS Between Two Images

modified algorithm, Hirschberg finally presents a divide-and-conquer algorithm to compute L. We implemented Hirschberg's algorithm and adapted it so that it accepts program images as input while preserving the $O(mn)$ time and $O(min(m, n))$ space complexities.

When the host system does not have access to the previous version of the program image or is not aware of the size of the previous image version, some subtle changes are made to the reprogramming strategy. When the host system receives a new program image of size n bytes to be programmed, it issues a read command to the boot loader executing on a sensor node in the network. The boot loader reads n bytes from application flash memory and returns the data back to the host. The host system treats these n bytes of data as the previous version of the program image. Thus, adapted to this scenario, the modified algorithm executes in $O(n^2)$ time and $O(n)$ space.

4.2.1.2 Edit Map Creation

Edit map creation is the second sub-phase of difference generation. The edit map is composed of two types of data. The first consists of the LCS segments. From a network reprogramming perspective, transferring the LCS segments from the host system to the sensor nodes would be redundant since that data already exists as part of the old program image. However, to reprogram a sensor node, the boot loader needs information about the starting addresses and extents of the LCS segments, along with the destination addresses where the segments need to be relocated. The second type of data consists of the new program image data segments that *do not* belong to the LCS and must be transferred to the sensor node.

The LCS segments are further classified into two sub-groups. The first sub-group consists of

data segments that reside in the same address locations in the old and the new program images; l_1, l_4, l_7, and l_9 in Figure 4.7 are examples. Since these data segments are already where they need to be, they are not included as part of the edit map. The second sub-group consists of data segments that need to be moved from one address location to another; l_2, l_3, and l_5 are examples.

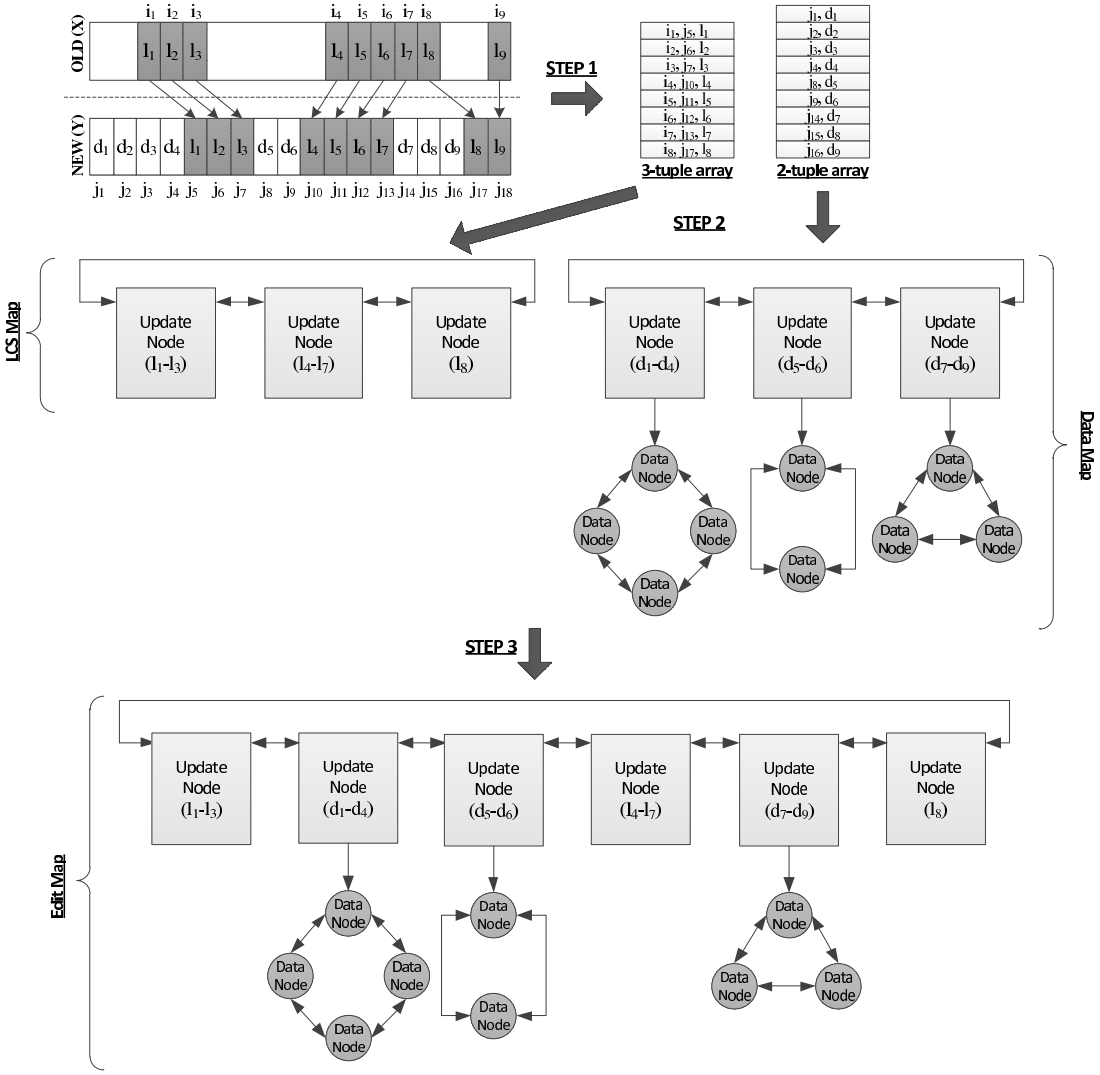


Figure 4.8: *Edit Map* Generation Flowchart

Map Generation: The output of the adapted algorithm is L, the string containing the longest common subsequence of X and Y. Figure 4.8 illustrates the steps involved in generating the edit map from the LCS. The first step involves calculating the locations of the LCS segments $l_1, l_2, l_3, ..., l_r$ in X and Y in $O(n)$ time and storing them as an array of 3-tuples, (i, j, l_k), where l_k

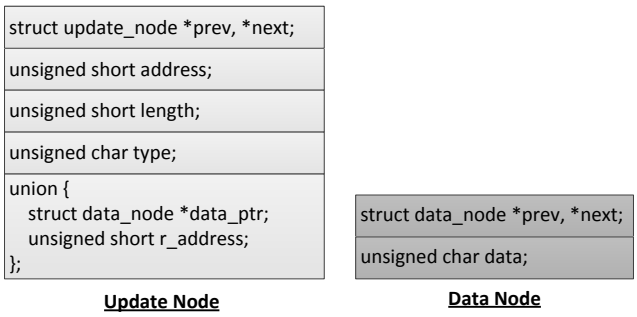


Figure 4.9: Update and Data Node Structures

is the k^{th} element of L, and i and j are its corresponding locations in X and Y, respectively. A 3-tuple entry is not created for cases where $i = j$, e.g. in the case of l_9 in Figure 4.8. Non-LCS data in Y (the new program image) is stored in an array of 2-tuples, (j, y_j), where j is the location of the data element and y_j is the j^{th} element. Both arrays are sorted by j.

The second step involves building an *LCS map* and a *data map*; the *update node* and *data node* structures used for creating these maps are shown in Figure 4.9. Both data structures consist of a doubly-linked list of update nodes. Each update node contains address, length, and type fields, as well as a union over a pointer to a data node and a remote address. Each data node contains one byte of data. The fields are considered in more detail in the remainder of the section.

The LCS map is built by scanning the 3-tuple array and combining LCS segments with contiguous X and Y address locations. Consider LCS segments l_4, l_5, l_6, and l_7 in Figure 4.8. The starting address locations (i_4, i_5, i_6, and i_7), as well as the destination address locations (j_{10}, j_{11}, j_{12}, and j_{13}) of these four LCS segments are contiguous. Thus, a single update node entry is created for the entire range in the LCS map. The update node contains the starting address of the LCS segment in Y (j_{10}), the starting address of the LCS segment in X (i_4), the length of the segment (4), and a type indicating that the node is being used for the LCS map. Note that the starting address of the LCS segment in X is stored in the `r_address` field within the union (The `data_ptr` field is used in update nodes within the data map). Each LCS map entry corresponds to data that needs to be read from one location and written to a corresponding location.

The data map is similarly created by scanning the 2-tuple array from the previous step. The non-LCS data segments which have contiguous destination address locations (in Y) are identified and merged into individual update nodes, e.g., d_1, d_2, d_3, and d_4, with address locations j_1, j_2, j_3,

and j_4 are merged into a single update node. The update node representing such a data segment in the data map contains the starting address of the segment in Y (j_1), the length of the segment (4), and a pointer to a doubly-linked list of data nodes. The data nodes store the individual bytes at each address location (d_1, d_2, d_3, and d_4). Each data map entry corresponds to contiguous data blocks that need to be written in the new program image.

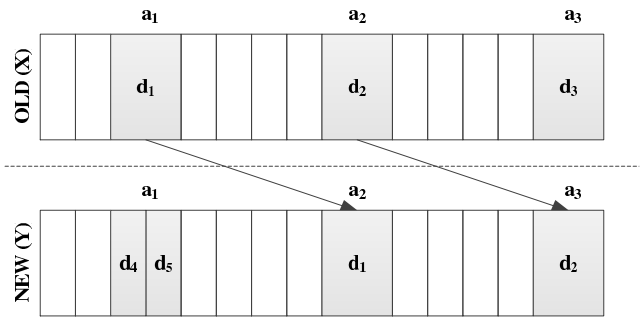


Figure 4.10: Update Ordering Problem

The final step involves merging the data map and the LCS map. The merge process is non-trivial, as it entails a priority ordered merge of the data and the LCS map elements. Incremental reprogramming requires data in flash to be moved in-place using limited RAM. Consider Figure 4.10, where the top and bottom images depict the state of flash memory before and after reprogramming, i.e., X and Y, respectively. The data segments in X at locations a_1, a_2, and a_3 are d_1, d_2, and d_3, respectively. Data segments d_1 and d_2 are part of the LCS, and hence need to be moved in memory, whereas d_4 and d_5 are new data segments which will be transmitted from the host system.

Assume that data segment d_1 is updated by incoming data segments d_4 and d_5 before it could be moved to location a_2, or that d_1 (in X) is moved to location a_2 (in Y) before d_2 is moved to location a_3. This would result in incorrect reprogramming. To avoid such scenarios, the order in which the data and LCS map elements need to be encoded in the edit map (and then transferred to the sensor nodes) is determined at the host system. The correct priority-based ordering in the scenario shown in Figure 4.10 is to move d_2, then move d_1, and then finally write d_4 and d_5. The ordering used while merging the data and LCS maps into the edit map prioritizes read operations at address locations in X over write operations at address locations in Y.

Map Encoding: The edit map resulting from the combination of the data and LCS maps is encoded for efficient transmission. The host system uses three instructions to encode the edit

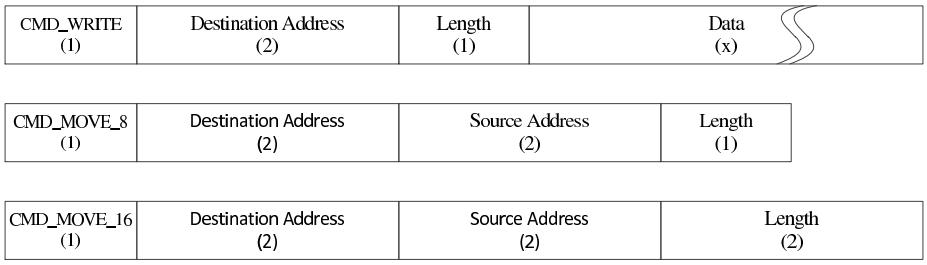


Figure 4.11: Edit Map Encoding Scheme

map: `CMD_WRITE`, `CMD_MOVE_8`, and `CMD_MOVE_16`, the formats for which are shown in Figure 4.11. Each of the operations require 1 byte to represent the op-code.

The write operation, `CMD_WRITE`, is used to represent each of the data map elements in the edit map, identified by the type variable in the update node structure in Figure 4.8. The write operation uses 2 bytes to specify the address location where the write should occur, 1 byte for the length of the data that must be written, and `x` bytes for the data itself, where `x` is the value of the length variable. The maximum amount of data that is transferred in a single message is set to 256 bytes; this allows the use of 1 byte for the length variable (by using the value 0 to represent 256).

The move operations, `CMD_MOVE_8` and `CMD_MOVE_16`, are used to represent each of the LCS map elements in the edit map. They both use 2 bytes each to specify the destination and source address for the LCS segments to be moved. `CMD_MOVE_8` is used to specify move operations for segments with length less than or equal to 256, and hence uses 1 byte for the length variable, while `CMD_MOVE_16` uses 2 bytes for longer segments.

Figure 4.12 illustrates a representative edit scenario. For the LCS segment l_1, a `CMD_MOVE_8` operation is used. A `CMD_MOVE_16` operation is used for the segment l_3, as the length of l_3 is greater than 256. Since LCS segment l_2 is already in its final position, it does not need an update. The remainder of the data segments in the image are populated using `CMD_WRITE` operations. Based on the priority ordering described previously, `CMD_MOVE_8` will be the first operation to be sent, followed by the remaining operations, from left to right, as in the figure.

Map Optimization: Optimization begins from the encoding phase and affects the map generation phase in an iterative fashion, as shown in Figure 4.6. Encoding a single `CMD_MOVE_8` operation requires 6 bytes, regardless of the length of the LCS segment being moved, as shown in Figure 4.11. Encoding a complete `CMD_WRITE` operation requires 4 + `x` bytes, where `x` is the length

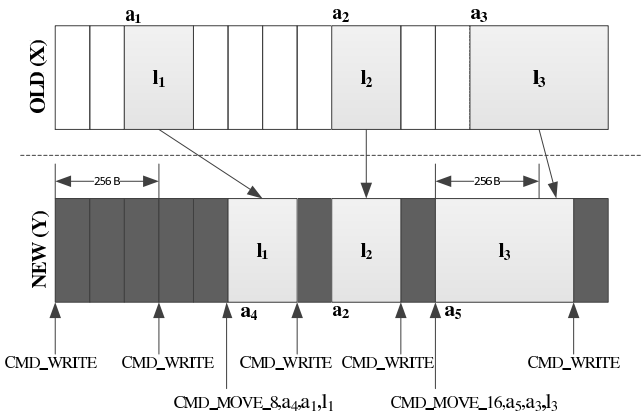


Figure 4.12: Application of Edit Operations

of the data segment being transmitted. A `CMD_MOVE_8` operation with a length value of 2 can be converted into a `CMD_WRITE` with no change in communication cost, while a move operation with a length value of 1 can be converted into a write operation requiring 1 byte less. Converting `CMD_MOVE_8` operations to `CMD_WRITE` operations also allows for multiple write operations to be consolidated into one when the segments are contiguous. The optimization step builds on this idea and uses it as a heuristic to reduce the encoded edit map size.

The first step in the optimization phase consists of calculating the cost of the incremental update, i.e., the number of bytes required to encode the edit map, denoted by $\mathtt{C_{update}}$. The update cost is calculated as follows:

$$\mathtt{C_{update}} = 6 * \mathtt{N_{lcs\_map\_8}} + 7 * \mathtt{N_{lcs\_map\_16}} + 4 * \mathtt{N_{data\_map}} + \sum_{i=1}^{\mathtt{N_{data\_map}}} \mathtt{L_i}$$

where $\mathtt{N_{lcs\_map\_8}}$ and $\mathtt{N_{lcs\_map\_16}}$ are the number of LCS map elements in the edit map which use `CMD_MOVE_8` and `CMD_MOVE_16` operations, respectively; $\mathtt{N_{data\_map}}$ is the number of data map elements in the edit map; and $\mathtt{L_i}$ is the length of the data segment contained within the $\mathtt{i^{th}}$ data map element.

In the next step, a copy of the edit map is saved, and the edit map is subjected to a *merge*. Merging involves converting all LCS map entries of size less than or equal to a specified *merge window* to data map entries, and then running a linear scan to consolidate newly formed contiguous data map entries. The value for $\mathtt{C_{update}}$ is recalculated post-merge and compared with its last known value. The merge window value is initially set to 2 for the first merge operation. In subsequent iterations, the edit map is merged, while the window value is incremented by powers of 2, and

C_{update} is recalculated. A binary search is employed to determine the merge window value for which C_{update} attains a minimum. Once this window has been determined, the saved edit map is subjected to a final merge operation and then encoded for transmission.

Map Validation: The encoded edit map that has been created in the previous phase is checked for errors before it is transmitted to the sensor nodes. Map validation is achieved by decoding the encoded edit map and applying the resulting operations to a copy of the older image version, X. After application of the edit map to X, the resulting image is compared to the desired new image version Y, using a linear scan in $O(n)$ time. A successful match validates the encoded edit map and initiates the data dissemination phase.

4.2.1.3 Data Dissemination

The data dissemination phase involves the generation of fixed-size data packets from the edit map and their subsequent transmission to sensor nodes using an XNP-like data dissemination protocol implemented in C. The boot loader provides the reprogramming logic and is also responsible for the reception of these packets. After the packets are received, the edit map is temporarily stored in external memory and node reprogramming is initiated.

4.2.1.4 Node Reprogramming

The node reprogramming phase consists of decoding the individual edit map operations and using these operations to reconstruct the new program image. The `CMD_WRITE` and `CMD_MOVE_8` operations are trivial to perform; typically data for a `CMD_WRITE` operation is already available, while the contents of a `CMD_MOVE_8` (up to 256 bytes) can be copied to RAM, and then moved to the new memory location. `CMD_MOVE_16` operations are slightly more complex to perform when the data to move is larger than the RAM capacity, and the starting and ending ranges overlap. Under such circumstances, our approach ensures that updates always occur without destruction of necessary data. An alternate approach is to use an external memory module as a buffer. In this scenario, data manipulation is performed in external memory, and then moved back to its new location in on-chip flash.

For our implementation of this approach, we use Ferroelectric RAM (FRAM) as the external memory module[1]. FRAMs are characterized by non-volatility, low power consumption (significantly

[1]Using an external FRAM memory module saves energy expended on flash reads, since flash writes occur at page

lower than flash memory), faster read and write performance (comparable to SRAM), and a higher number of write-erase cycles [54]. Additionally, FRAMs provide byte addressable memory, like NOR flash devices.

The first time a sensor node is programmed using an ISP, the program image is also written to the FRAM device. When the edit map is received by the sensor node, the initial data manipulation is done using the FRAM. At the end of the image reconstruction phase, the FRAM contains the updated version of the image, which is then written to on-chip flash memory. This process ensures that when the sensor node is not being reprogrammed, the image in flash is mirrored in FRAM.

4.3 Evaluation

4.3.1 Experimental Setup

We implemented the incremental code update reprogramming strategy for the MoteStack [22], a state-of-the-art in-situ sensing platform, which uses an AVR Atmel (ATMega 644P) microcontroller (MCU) operating at 10 MHz and powered at 3.3V. The MCU consists of 64KB of in-system-programmable flash memory, 2KB of EEPROM, and 4KB of SRAM. The boot loader is installed in on-chip flash, which offers read-while-write capabilities. We added a 64KB FRAM memory device [54] to use for the image reconstruction phase.

We consider five software change scenarios involving the latest stable version of our custom C-based sensor operating system (with standard OS services) as test cases for our evaluation.

1. **Changing a constant (minor change).** We use a standard blink application as our base case and change a constant to make the LED blink every two seconds (instead of one).

2. **Modification of implementation file (moderate change).** We add 91 lines of (non-whitespace) code to convert the base application into a LED test suite, where various patterns are displayed on five LEDs.

3. **Changing an installed application (major change).** We next write an application to manipulate external flash memory. The new application writes a data buffer filled with random data to external flash memory, and then reads the page back.

granularity.

4. **Modification of core OS (moderate change).** We next comment out a few lines of code so the new application version does not contain the ZigBee driver module.

5. **Modification of core OS (moderate change).** We next comment out a few lines of code so the new application version does not contain the Wi-Fi driver module.

We also consider five scenarios to evaluate how the approach performs when applied to typical code changes in TinyOS, using standard applications from the `apps` directory of the TinyOS 2.1.0 code distribution[1]. Comparisons are made between our code update strategy and two other state-of-the-art reprogramming strategies, Zephyr [48] and Rsync [65]:

6. **Changing a constant (minor change).** We change a constant in the `Blink` application to alter the LED blink rate and reprogram a basic `Blink` application install.

7. **Changing an installed application (major change).** We next replace the installed `Blink` application with the `RadioCountToLeds` application.

8. **Modification of implementation file (moderate change).** We next comment out a few lines of code from `RadioCountToLeds` to emulate a moderate code change.

9. **Modification of core OS (moderate change).** We next comment out a few lines of code from `RadioCountToLeds` to remove the `AMControl` module responsible for radio communication.

10. **Modification of core OS (moderate change).** Finally, we comment out a few lines from `RadioCountToLeds` to remove the `Leds` module.

4.3.1.1 Data Transmission Savings

We first evaluate the performance of the incremental code update approach. The percentage compression in data size to be transmitted when reprogramming a node (P_{tx}) is calculated as the ratio of the length of the generated edit script ($\mathtt{C_{update}}$) to the length of the new program image version (L_{new}), given as $\mathtt{C_{update}} * 100/L_{new}$. It is expressed as a percentage, reflecting the percentage of data that is transmitted using the incremental update approach compared to transmitting the full image. The packet overhead during data dissemination is dependent on packet length and protocol; it is not considered as part of the evaluation of the edit script generation strategy.

[1] Code change scenarios 6-10 are replicated from [48].

Device Type	Read	Write	Erase
NAND Flash	2.29 nJ/B	14.55 nJ/B	4.03 nJ/B
NOR Flash	2.09 nJ/B	793.75 nJ/B	881.25 nJ/B
FRAM [54]	0.33 nJ/B	0.33 nJ/B	-NA-

Table 4.1: Energy Consumption Characteristics

4.3.1.2 Merge Window Optimization

We next evaluate the effect of varying the merge window size on the size of the generated edit map while using the custom C-based sensor OS. We record the $\mathtt{C_{update}}$ values for fixed merge window sizes of 0 (indicating that a merge will not be conducted), 1, 2, 4, 8, 12, and 16, for cases 1-5. A merge window of w bytes converts LCS map entries of length less than or equal to w into data map segments, so as to consolidate the newly formed contiguous data map entries into a single entry.

4.3.1.3 Image Reconstruction Cost

Finally, we evaluate the cost of image reconstruction on the sensor node and consider the potential energy savings. Let `h` be the length of the program image, `m` be the amount of data that needs to be moved in FRAM (due to `CMD_MOVE_8` and `CMD_MOVE_16` operations), and `w` be the amount of new data that needs to be written in FRAM (due to `CMD_WRITE` operations). Let $\mathtt{R_{FRAM}}$ and $\mathtt{W_{FRAM}}$ be the cost of reading and writing a byte of data in FRAM, respectively. Finally, let $\mathtt{W_{FLASH}}$ be the cost of writing a byte in flash, and $\mathtt{C_t}$ be the cost of transmitting a byte wirelessly. The cost of simple reprogramming, $\mathtt{C_r}$, is given by[1]:

$$\mathtt{C_r} = \mathtt{h} * (\mathtt{C_t} + \mathtt{W_{FLASH}})$$

whereas the cost of incremental reprogramming with image reconstruction, $\mathtt{C_i}$, is given by[2]:

$$\mathtt{C_i} = \mathtt{C_{update}} * \mathtt{C_t} + \mathtt{m} * (\mathtt{R_{FRAM}} + \mathtt{W_{FRAM}}) + \mathtt{w} * \mathtt{W_{FRAM}} + \mathtt{h} * \mathtt{W_{FLASH}}$$

Table 4.1, adapted from [49, 36, 54], presents the energy consumption characteristics of NAND flash, NOR flash, and FRAM devices for read, write, and erase operations. For wireless

[1]The cost of protocol and control flow data is not considered.
[2]The cost of executing instructions on the MCU is not considered.

	Case 1	**Case 2**	**Case 3**	**Case 4**	**Case 5**	**Case 6**	**Case 7**	**Case 8**	**Case 9**	**Case 10**
L_{old} (B)	37594	37594	38068	38294	37674	2650	2650	11526	11512	11378
L_{new} (B)	37594	38068	38294	37674	35276	2650	11526	11512	11378	11220
$\mathtt{C_{update}}$ (B)	5	4966	9748	2042	2815	6	10241	1121	1194	1173
P_{tx}	0.013%	13.05%	25.45%	5.42%	7.98%	0.22%	88.5%	9.74%	10.49%	10.46%

Table 4.2: Edit Map Sizes for C-based OS and TinyOS Update Scenarios

data transmission, we consider the XBee low power RF module [30]. Assuming that the XBee module performs at the maximum advertised rate of 250,000 bps, the transmit and receive energy requirements are 4.75 mJ/B and 5.8 mJ/B, respectively. We use the average of the transmit and receive costs, and set $\mathtt{C_t}$ to 5.28 mJ/B.

4.3.2 Results

We analyze the results of the experiments in detail in the context of data transmission savings, merge window optimizations, and the cost of image reconstruction.

4.3.2.1 Data Transmission Savings

Table 4.2 presents the edit map sizes (C_{update}) and P_{tx} values achieved for both the C-based OS and TinyOS code update scenarios, along with the corresponding old and new image sizes (L_{old} and L_{new}, respectively). In case 1, the difference between the two program images is small, and this is reflected in the small $\mathtt{C_{update}}$ size (5 bytes) and the correspondingly small P_{tx} (0.013%). In case 2, which is a typical change in the software development life cycle of an embedded device, the edit map size is 4966 bytes, resulting in a P_{tx} of 13.05%.

In case 3, which involves the most significant code change, the edit map size is 9748 bytes, resulting in a P_{tx} of 25.45%. This can be explained by the fact that even though the *Blink* and *Flash* applications are very different, they share approximately 88% of the base OS code ($100 - \mathtt{w} * 100/L_{new}$). In cases 4 and 5, the P_{tx} values achieved are 5.42% and 7.98%, respectively. The smaller edit map sizes, 2042 bytes and 2815 bytes, respectively, for cases 4 and 5, can be attributed to the fact that even though entire functions are removed in both changes, the LCS-based approach correctly accounts for the shifts in the other functions within the code image.

In scenarios involving minor to medium code changes in TinyOS, cases 6 and 8, the approach achieves relatively small edit map sizes of 6 and 1121 bytes, respectively, and correspondingly small P_{tx} values of 0.22% and 9.74%, respectively. Case 7 involves a major change, where the `Blink`

Merge Window	Case 1	Case 2	Case 3	Case 4	Case 5
0	5	5033	12436	2097	3173
1	5	5013	12080	2056	3039
2	5	4979	10810	2056	2829
4	5	4966	10036	2042	2825
8	5	5051	9750	2044	2815
12	5	5190	9986	2046	2832
16	5	5424	10471	2052	2888

Table 4.3: Edit Map Sizes for Different Merge Windows in Cases 1-5

application is updated to `RadioCountToLeds`. The large edit map size (10241 bytes) and P_{tx} value achieved (88.5%) is due to the difference in size between L_{old} and L_{new}. Specifically, 8876 bytes of new data must be transferred; only 1365 bytes ($10241 - 8876$) are transferred to rebuild the new image. The P_{tx} values in cases 9 and 10 (10.49% and 10.46%, respectively) can again be attributed to the ability of the LCS-based approach to account for function shifts.

4.3.2.2 Merge Window Optimization

Table 4.3 summarizes the impact of varying the merge window size on the size of the resulting edit map; the results are plotted in Figure 4.13. In each graph, the horizontal axis represents the merge window size, and the vertical axis represents the cost of update ($\mathtt{C_{update}}$). Case 1 is unaffected by window size; this is due to the fact that it consists of a single `CMD_WRITE` operation of length one, and there is nothing to merge. In the remaining cases, the minimum $\mathtt{C_{update}}$ value is achieved at or beyond a merge window value of 2. (Encoding a 2-byte `CMD_MOVE_8` costs the same as a 2-byte `CMD_WRITE`.) We observe that case 3 has the highest rate of change as a function of window size, followed by cases 2, 5, 4, and 1, corresponding to their overall $\mathtt{C_{update}}$ costs. A larger $\mathtt{C_{update}}$ cost indicates a higher degree of dissimilarity between two program images, as well as higher fragmentation in the LCS segments. The more fragmented the LCS segments, the higher the chances of merging multiple data segments. This often makes the *map optimization* strategy more effective for cases with higher $\mathtt{C_{update}}$ values (major code updates).

4.3.2.3 Image Reconstruction Cost

Table 4.4 compares the cost of simple reprogramming ($\mathtt{C_r}$) with the cost of incremental reprogramming ($\mathtt{C_i}$), the results of which are illustrated in the bar plot in Figure 4.14. We observe that the ratio of the cost of simple reprogramming to the cost of incremental reprogramming strongly

	Case 1	Case 2	Case 3	Case 4	Case 5	Case 6	Case 7	Case 8	Case 9	Case 10
`m` (Bytes)	0	24766	27873	1996	3492	0	1344	7279	7178	7292
`w` (Bytes)	1	1772	4572	830	1389	2	9875	437	530	503
C_r (Joules)	198.497	201	202.193	198.919	186.258	13.992	60.857	60.783	60.076	59.242
C_i (Joules)	0.0269	26.221	51.47	10.782	14.864	0.0317	54.073	5.919	6.304	6.194

Table 4.4: Comparison of Simple and Incremental Reprogramming Costs

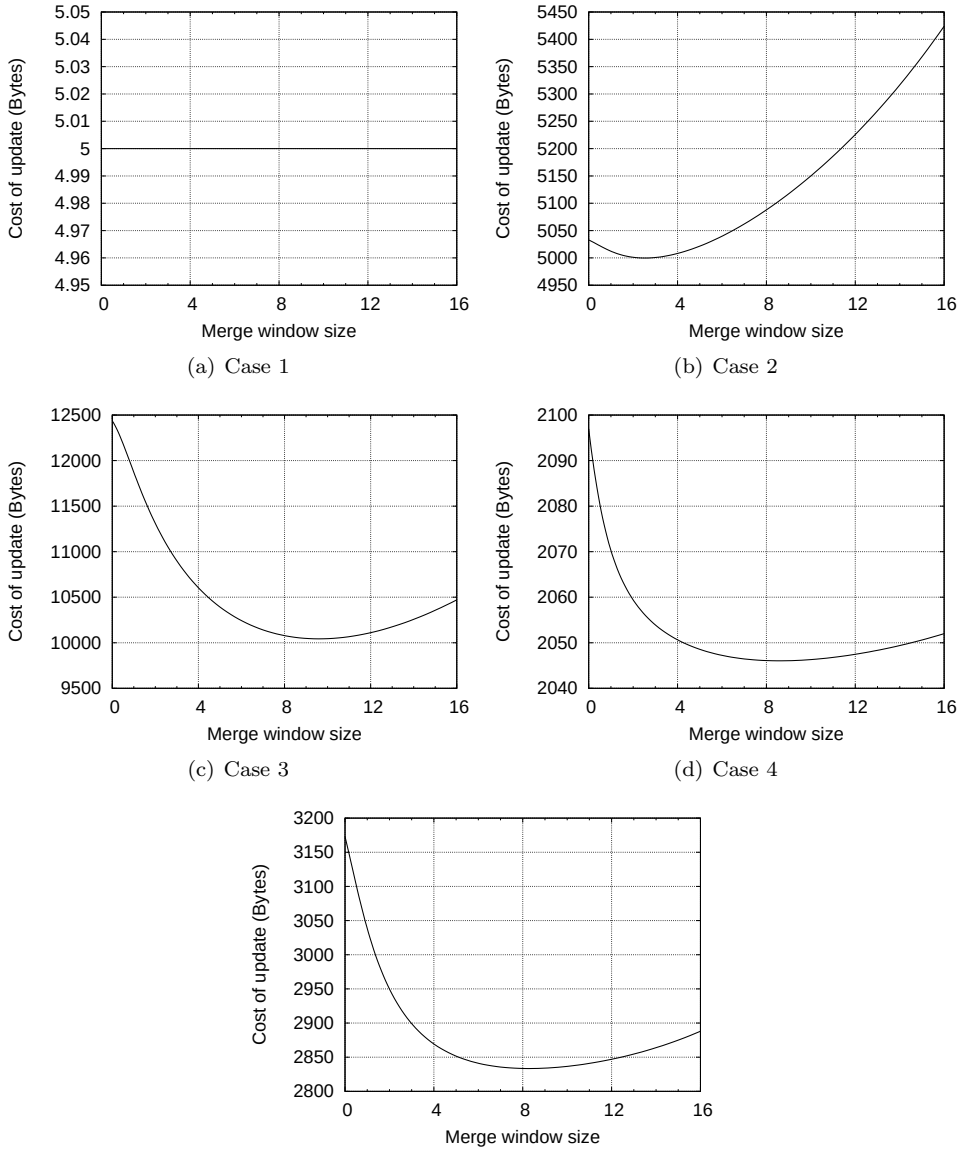
correlates to the P_{tx} values achieved in Table 4.2, in spite of the wide variation in the number of bytes of data moved (`m`) and written (`w`) in FRAM. This can be attributed to the fact that the amount of energy expended in transmitting a byte of data is about 16,000,000 times more than the amount of energy expended to write a byte of data in FRAM, and about 350,000 times more than writing a byte of data to flash. Thus, a higher degree of data transmission savings, even at the expense of flash and other external memory manipulation, results in a lower energy footprint.

Panta et al. report detailed comparisons of delta script sizes for various incremental reprogramming approaches in [48]. Zephyr, when using application-level modifications, reduces the data size to be transmitted to 0.07% for minor changes, and between 1.18% and 23.31% for moderate code changes. Without the application-level modifications, Zephyr reduces the data size to 0.096% for minor code changes, and between 9.17% and 36.63% for moderate code changes. Rsync reduces the size of data to be transmitted to 2.56% for minor code changes, and between 24.47% and 45.65% for moderate changes. In comparison, our approach achieves comparable (and often better) reductions in transmitted data size and resulting energy savings for similar code update scenarios. Unlike these approaches, however, our approach does not assume any knowledge of program code structure. It is platform and language independent.

4.4 Summary

We first presented the design and implementation of VSPIN, a Linux-based framework for developing and testing incremental code update mechanisms. Next, we described an incremental code update mechanism for efficient wireless sensor network reprogramming. Our approach uses an adaptation of Hirschberg's Algorithm to generate an edit script based on the differences between two program images. We use a heuristic-based optimization strategy to reduce the edit script size, which is then transmitted to sensor nodes using a standard data dissemination protocol. Finally, the sensor nodes decode the edit script and use it to construct the new program image. The approach reduces

the required transmission data size to 0.013% for minor changes, and between 5.42% and 13.05% for moderate code changes when using a custom C-based sensor OS. When applied to TinyOS, the approach reduces the data size for transmission to 0.22% for minor changes, and between 9.74% and 10.49% for moderate code changes. Our approach compares favorably to (and often better than) prior work in the area. The reduction in the amount of data needed to be transmitted leads to significant energy savings for wireless sensor network reprogramming. At the same time, our approach is platform and programming language independent and assumes no knowledge of program code structure.



(a) Case 1

(b) Case 2

(c) Case 3

(d) Case 4

(e) Case 5

Figure 4.13: Effects of *Map Optimization* on $\mathtt{C}_{\mathtt{update}}$

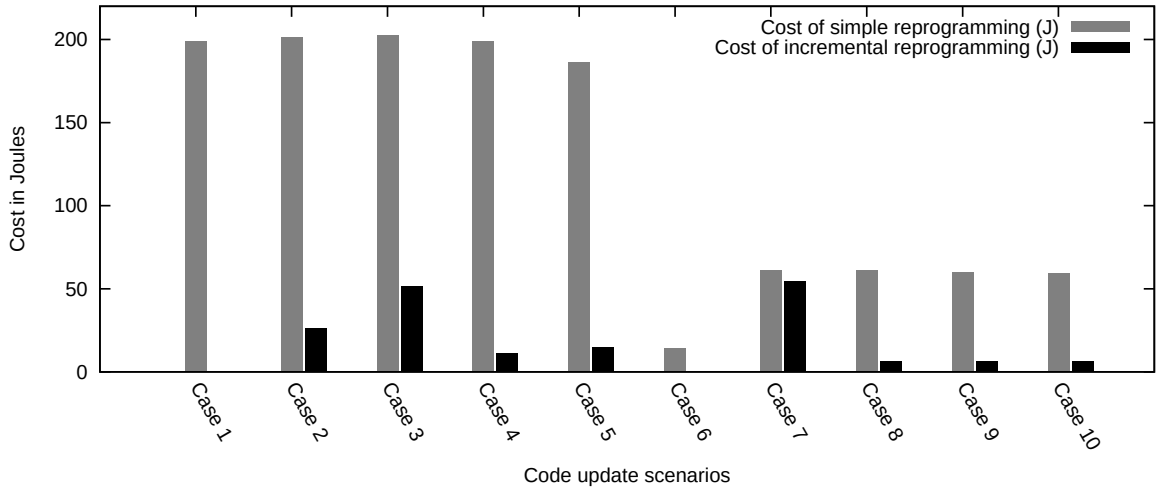


Figure 4.14: Reprogramming Costs for Different Code Update Scenarios

Chapter 5

Data Storage

In this chapter, we present the design and implementation of LoggerFS, a fast and reliable data storage solution for WSNs. We present a consistent data structure update mechanism used to achieve data consistency and file system reliability. We also present the design of a FRAM-based, write-back cache which allows for efficient write-batching and flash wear-leveling. We demonstrate the fault tolerance capabilities of LoggerFS and evaluate the read and write performance of a prototypical LoggerFS implementation.

5.1 LoggerFS Design

LoggerFS is optimized to address the design considerations discussed in Section 2.3, including the identified data and file access patterns. Figure 5.1 illustrates the hardware architecture of a sensor node running LoggerFS. The sensor node's MCU includes integrated RAM and EEPROM, both limited in size. We extend these facilities with an external FRAM module and an external flash module.

Data generated by attached sensors $\{S_1,...,S_n\}$ is processed by the MCU and ultimately written to flash. The data is later read from flash and transmitted to a base station using a wireless module. Alternatively, binary program image data (or the image *diff*) is received by the wireless module from the base station and stored in flash memory. The data is later read from flash, and the program image is reconstructed and written to the programmable flash memory of the sensor node [45, 31, 32]. Configuration data is stored as a special file in external flash.

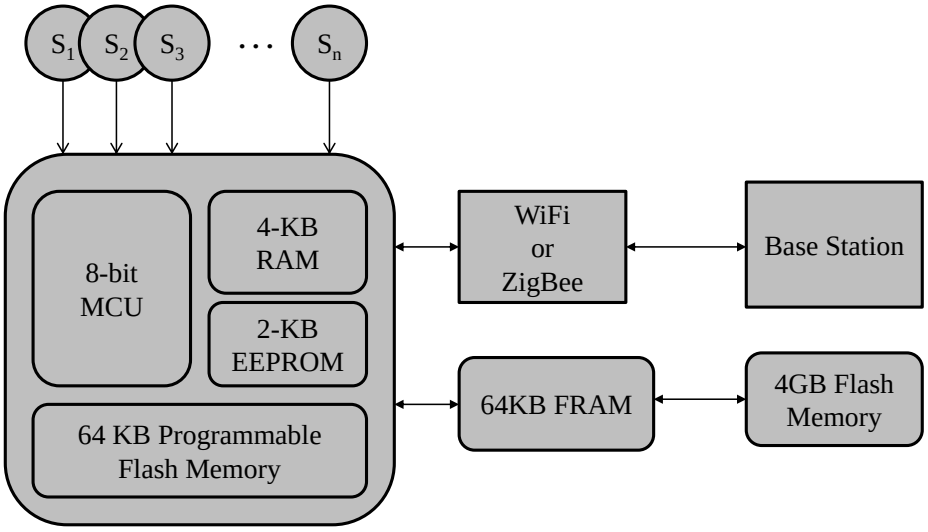


Figure 5.1: LoggerFS Hardware Architecture

Figure 5.2 summarizes the content structure across the three memory modules in the hybrid storage design. Files are stored in flash memory. LoggerFS uses a flat structure since the total number of files in an embedded system is expected to be low. The structure supports fast lookup and update mechanisms.

During execution, each file is represented in primary memory (RAM) by a metadata record. Updates to a file require updates to the metadata, both in volatile and non-volatile memory. As discussed in Sections 2.3.1 and 2.3.2, there are challenges associated with writing data to flash memory. Smaller updates, such as file metadata, tend to be more problematic than large block updates, due to the page-sized write granularity of flash memory. Consequently, LoggerFS stores file metadata in FRAM. Since FRAM allows for fast, byte-sized write operations, frequent updates to the metadata do not pose a challenge.

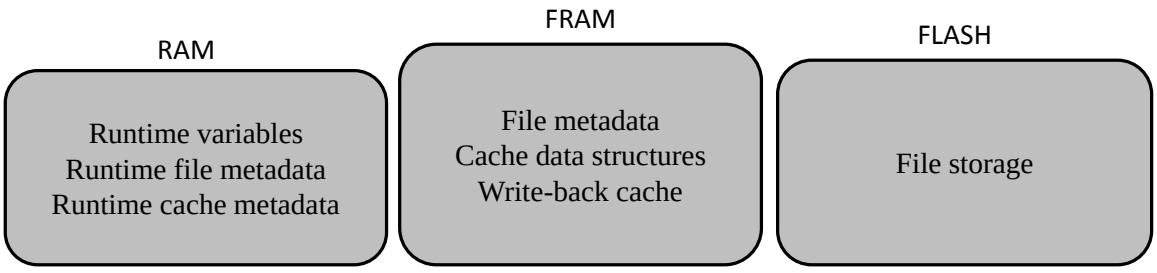


Figure 5.2: Hybrid Storage Approach

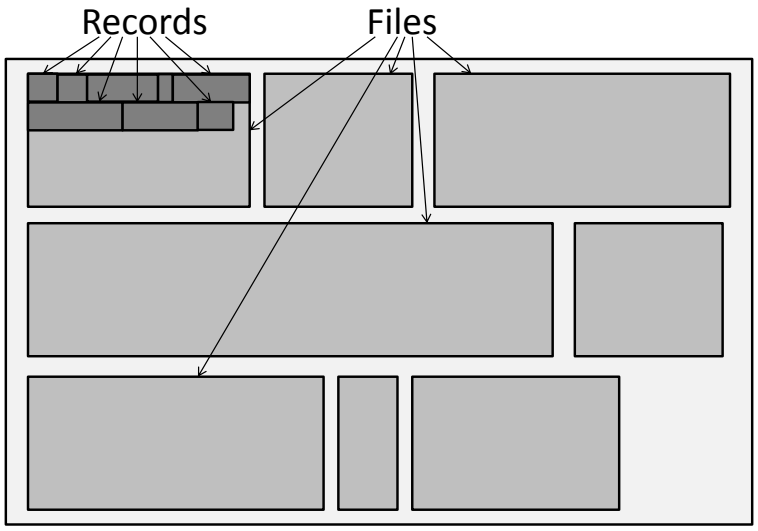


Figure 5.3: Logical View of File and Record Layout

FRAM storage is partitioned into two parts, a metadata store and a write-back cache. In addition to storing the file metadata, the metadata store maintains the data structures required to support the buffer cache, which are loaded in primary memory during execution. The write-back cache is used to facilitate fast write caching in FRAM during periods of high-rate sensing. The size of the write-back cache is significantly larger than the free space available in RAM, thus allowing for efficient write-batching and flash wear-leveling.

The runtime file metadata (associated with files being accessed), the variables required to maintain the file metadata (e.g. stack variables and pointers), and cache data structures are stored in RAM. To prevent the loss of file system state due to the loss of runtime variables stored in RAM (e.g., as a result of unexpected power loss), LoggerFS uses a lightweight *consistent data structure* to achieve data persistence. We discuss the data structure in Section 5.1.3.

5.1.1 Files in Flash

Figure 5.3 illustrates the logical view of file and record layouts in flash. Sensor nodes are primarily used to log sensed data, and to transmit this data to a base station. New files are usually not created once a system has been deployed. For these reasons, LoggerFS supports a fixed number of files, configured during system installation. Each attached sensor is paired with a file using a unique identifier; the maximum number of files supported depends on the size of external flash

memory and the configuration of file sizes. File sizes are configured at installation time based on the sensing rate and maximum required critical data period. Data logged by a sensor device during a single sampling period is called a *record*.

Each file in LoggerFS is a circular buffer capable of containing variable-sized records. The update granularity is one record. New records are always appended to the end of a file, while old records are always read and/or deleted from the beginning of a file. If a file is full, and the node is unable to communicate with the base station, new records overwrite old records.

Files in flash memory are block-aligned to account for flash write and erase constraints. Updates to a file always occur through the write-back cache resident in FRAM. This ensures efficient batching of writes and reduces flash wear, discussed in Section 5.1.4.

5.1.2 File Metadata

Files are represented in primary memory (and in FRAM) as file metadata. LoggerFS uses constant-sized metadata to represent each file; the size does not increase with file size. Since the number of files is preconfigured, the amount of space required to store the file metadata in FRAM is known during installation. This allows LoggerFS to assign space for the write-back cache after accounting for the space required for the associated data structures.

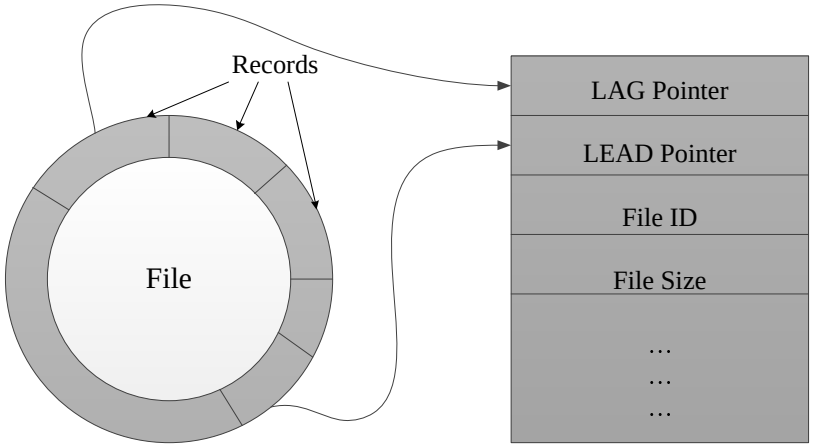


Figure 5.4: File Metadata in RAM and FRAM

Figure 5.4 illustrates the metadata associated with each file. A pair of pointers, `LAG` and `LEAD`, are used to maintain the circular files. The `LAG` pointer points to the oldest record, while the `LEAD` pointer points to the newest record. The size of the record is contained within the record itself,

allowing minimal information to be stored in metadata, in addition to supporting variable-sized records. Additional information, such as the file identifier and file size, are stored as part of the file metadata to support fast file access and update.

5.1.3 File System Consistency

A lightweight, consistent data structure is used by LoggerFS to guarantee file system consistency. Every variable-sized record is first converted to a sequence of writes, the last of which is an *atomic* write operation. Consider Figure 5.5, which illustrates the idea behind the consistent data structure. All updates to a file in LoggerFS occur through FRAM via the write-back cache. Each variable-sized block of data (record) written in FRAM is converted to a pointer update. The pointer contains the starting address of the record and is only updated after the block of data has been successfully written to FRAM. A one-bit *flip* pointer index is finally updated after the pointer address has been successfully recorded in FRAM.

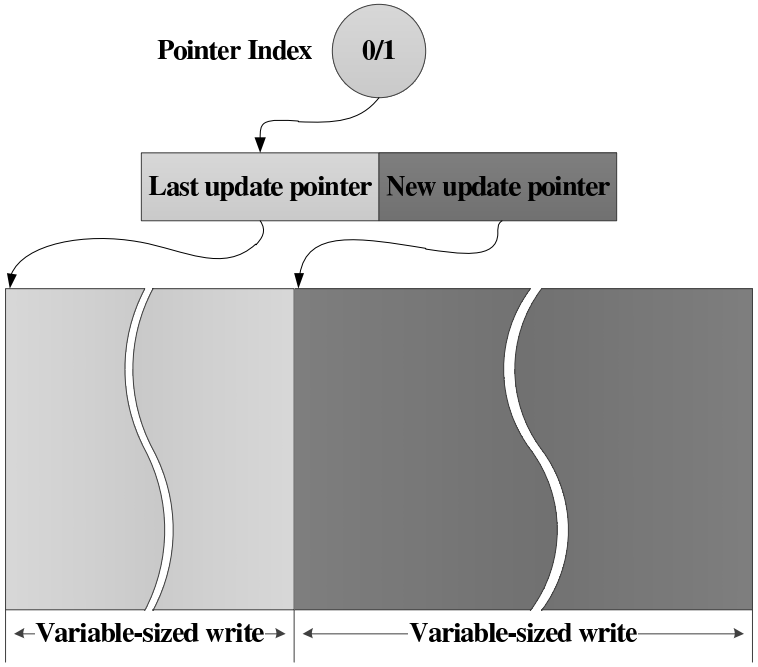


Figure 5.5: Consistent Data Structure Update

Consider an example. Let the state of the data structure be such that the lightly shaded block in Figure 5.5 is the last known correctly saved record in FRAM. The lightly shaded *last*

update pointer contains the starting address of this block of data, and the *pointer index* records a `0`, indicating that the *last update pointer* is the last known valid value of the pointer. Assume that a new block of data must be written, indicated by the darkly shaded block of data in the figure. After the darkly shaded block of data is written to FRAM, the *new update pointer* is updated. Following the update, the pointer index is flipped to a `1`. Since the update of a single bit is an atomic operation in any architecture, the entire block update is thus logically converted to an atomic update. A write operation failure at any point before the pointer index bit update does not corrupt the previous valid record entry.

Reading a record from this structure involves reading the pointer index bit, looking up the corresponding pointer value, and then reading the corresponding block of data using the address stored in the pointer.

Since all data structures used in LoggerFS are consistent, and every write is atomic, we are able to achieve file system consistency without needing any additional software mechanisms, e.g., checkpointing/consistency points.

5.1.4 FRAM-based Write-back Cache

The FRAM-based write-back cache allows LoggerFS to handle bursts of high-rate sensor data and alleviates wear on the flash module. The cache is again structured as a circular buffer capable of holding variable-sized files. Caching a file involves moving the file from flash memory to the cache; space for the entire file is allocated in the cache. A pair of LAG and LEAD file pointers are used to point to the least recent and the most recent files which were paged in from flash memory, respectively. The files in turn contain data records maintained by the LAG and LEAD pointers explained in Section 5.1.2.

Figure 5.6 presents the read workflow for LoggerFS. File read operations can be served from either cache or flash, depending on where a file is located. If a read operation arrives after a file has been cached for writes or deletes, the read can be served from cache. Read operations for an already evicted file are served from flash, but the read data (or file) is not cached. This simplifies cache manipulation because all cached files are always *dirty* and can be flushed to flash when evicted; no additional dirty bits need to be maintained.

Figure 5.7 presents the write workflow. All updates to a file occur through the write-back cache. When a record needs to be updated, the existence of the file is first verified in cache. If the

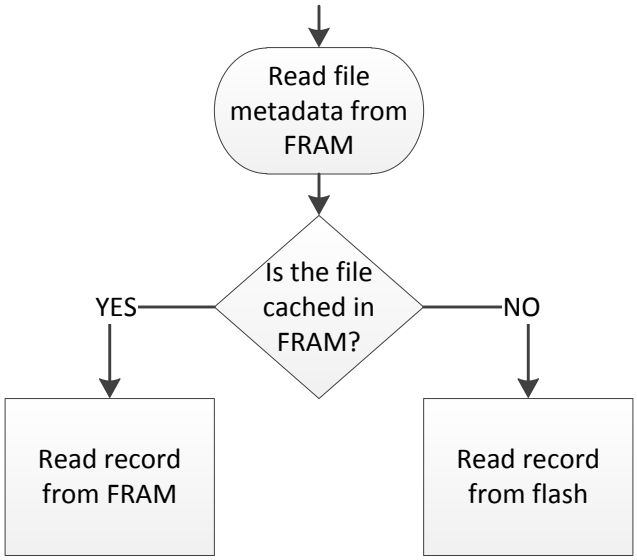


Figure 5.6: Read Path with Cache

file is already cached, the data record is updated in cache. If the file is not cached, the contents of the entire file are first paged into FRAM from flash before the record is updated.

5.1.4.1 Cache Eviction

Cache memory tends to get fragmented over time and usually needs sophisticated data structures to be managed. Since our target devices are resource-constrained, we use a FIFO file eviction policy, which eliminates in-memory cache fragmentation. Since the cache itself is a circular buffer, newly cached files are always written to the end of the buffer. Memory for the new file is allocated immediately after the most recent file entry in FRAM. If this new allocation causes the file to extend beyond the least recent file entry, the least recent file(s) is/are evicted until there is enough space to store the new file. File sizes are therefore limited by the size of FRAM.

5.1.4.2 Cache Data Persistence

Write operations to FRAM are faster than writes to flash (Table 1.1), making FRAM a strong choice for a write cache. Additionally, the size of FRAM is typically much larger than the size of RAM, at a fraction of the cost. Using FRAM for caching enables LoggerFS to buffer large amounts of data during high rate data bursts. To ensure reliability, records written to cache use the consistent data structure approach described in Section 5.1.3. As non-volatile memory, FRAM

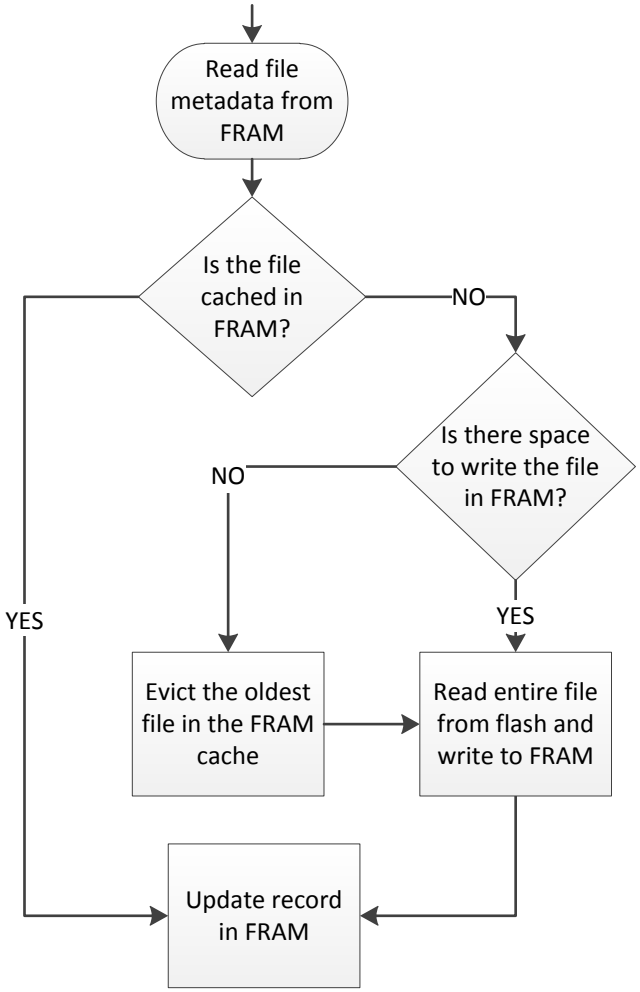


Figure 5.7: Write Path with Cache

allows all data stored in the cache to be available across device failures without the need for cache rewarming.

5.2 File System Implementation

We implemented LoggerFS in C for the MoteStack sensing platform [22]. The MCU, an ATMega644 [8], includes 64KB of in-system-programmable flash memory, 2KB of EEPROM, and 4KB of SRAM. Two external memory modules, a 64KB FRAM [55] and a 512KB flash [3], were added to support the file system. The following sections detail the file system implementation.

```
typedef struct record_header {
    uint16_t record_size;
    uint16_t event_id;
    uint8_t record_crc;
} record_header_t;

typedef struct fs_header {
    uint16_t file_id;
    uint16_t file_size;
    uint16_t fs_event_id;
    uint16_t lag_record_ptr;
    uint16_t lead_record_ptr;
    uint16_t cache_start_address_ptr;
    uint16_t flash_start_address_ptr;
    bool in_fram;
} fs_header_t;

typedef struct cache_header {
    uint16_t lag_file_ptr;
    uint16_t lead_file_ptr;
} cache_header_t;
```

Listing 5.1: File, Record, and Cache Metadata

5.2.1 File, Record, and Cache Implementation

Files in LoggerFS comprise multiple records. Listing 5.1 shows the representation structure of a record header (`record_header`), a file header (`fs_header`), and a cache header (`cache_header`). Record headers are included at the start of every record within a file and contain the size of the corresponding record. They also store a monotonically increasing event identifier (`event_id`) used to support file system consistency checks, and a checksum (`record_crc`). The `event_id` is used to verify continuity of update operations after a system is restarted following a crash and prevents further data corruption in the event of an unrecoverable hardware error.

File headers contain a file identifier (`file_id`), the size of the file (`file_size`), and a monotonically increasing file system event identifier (`fs_event_id`) used to support file system consistency checks. File headers store the LAG pointer (`lag_record_ptr`) and the LEAD pointer (`lead_record_ptr`), which contain logical offset locations. The physical record locations are calculated by adding the logical offsets (`lag_record_ptr` or `lead_record_ptr`) to the starting address of the file in cache (`cache_start_address_ptr`), or the starting location of the file in flash memory (`flash_start_address_ptr`). The starting address in flash memory is stored as a block address since files are stored block-aligned to accommodate flash erase granularity. The `in_fram` variable

```
1  bool         lfs_create(uint16_t config_record_size, uint8_t files, ...);
2  bool         lfs_reinit();
3  bool         lfs_write(uint8_t file_id, uint8_t *buf, uint16_t size);
4  uint16_t     lfs_read(uint8_t file_id, uint8_t *buf, uint16_t size);
5  uint16_t     lfs_read_verify(uint8_t file_id, uint8_t *buf, uint16_t size);
6  bool         lfs_delete(uint8_t file_id);
7  void         lfs_read_fs_hdr(fs_header_t *fs_hdr);
8  void         lfs_write_fs_hdr(fs_header_t *fs_hdr);
9  uint8_t      read_from_config(uint8_t index, uint8_t *value, uint8_t size);
10 bool         write_to_config(uint8_t index, uint8_t *value, uint8_t size);
```

Listing 5.2: LoggerFS API

indicates whether a file is present in FRAM cache or in flash memory. Since files can exist in both FRAM and flash at the same time, both `cache_start_address_ptr` and `flash_start_address_ptr` contain values.

The cache header also contains a LAG file pointer (`lag_file_ptr`) and a LEAD file pointer (`lead_file_ptr`) to keep track of contiguous space being used in the circular cache. The pointers point to the oldest and newest cached files, respectively.

5.2.2 File System API

Listing 5.2 presents the API exposed by LoggerFS. A new file system is created by calling `lfs_create`(). LoggerFS currently supports a single configuration file which may contain multiple configuration elements. The arguments to `lfs_create`() consist of the size of the configuration data file (`config_record_size`), the maximum number of supported files (`files`), and the size of each file (...), in bytes. A call to `lfs_create`() initializes FRAM and FLASH. Next, the configuration file size, number of data files, and corresponding file sizes are stored at the start of FRAM. Finally, the file metadata sections for all files are instantiated in FRAM.

The consistent data structure described in Section 5.1.3 is used to store the file metadata section in FRAM. Two copies of each file metadata entry are maintained; one copy contains the state of the file after the most recent update, while the other copy contains the state of the file before the update. Figure 5.8 illustrates the use of the consistent data structure to maintain consistent copies of the file metadata. The `new update pointer` and `last update pointer` point to the starting addresses of the new and old metadata entries, respectively. Consistency of record updates is safeguarded by using two copies of LEAD and LAG pointers, respectively stored in two copies of

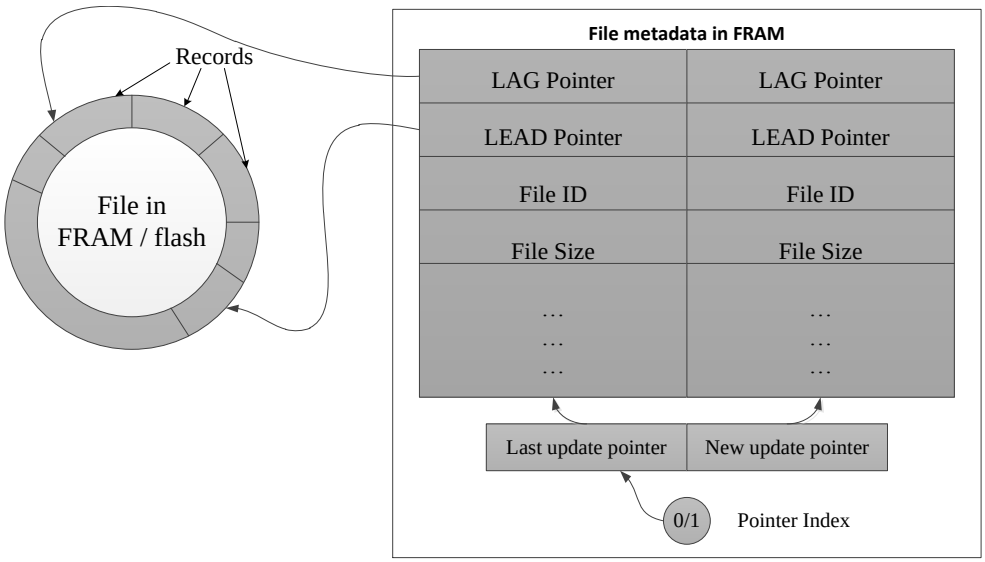


Figure 5.8: Consistent File Metadata

the file metadata.

A `magic` number is also stored at a fixed location in FRAM. During system initialization or restart, `lfs_reinit`() checks for the existence of a previously installed LoggerFS instance by checking for the presence of this `magic` number in FRAM. If a file system exists, the file sizes are read from FRAM, and the necessary run-time variables are loaded in to memory. The file and record metadata, which is also stored in FRAM, can then be accessed by the file system.

`lfs_write`() takes a file identifier (`file_id`) as argument, the data buffer (`buf`), and the size of the data buffer (`size`). A write entails reading the file metadata in `fs_hdr` via a call to `lfs_read_fs_hdr`() to identify the `LAG` and `LEAD` pointers. Using these pointers, the `LAG` and `LEAD` records are identified, and the correct write location in FRAM is calculated. Once the record is written, the `LAG` and `LEAD` pointers are recalculated and written back as part of the `fs_hdr` in FRAM via a call to `lfs_write_fs_hdr`(). The `LAG` pointer and corresponding record may need to be manipulated since write operations may trigger record deletions, if the file is full. The `LAG` pointer update is performed via `lfs_increment_lag_pointer`(), whereas the `LEAD` pointer is updated via the `lfs_write`() function itself, as part of the data write.

The `lfs_read`() and `lfs_read_verify`() functions are used to read the oldest record in a file. `lfs_read_verify`() additionally computes a checksum over the read data and compares it to the checksum stored in the record. The checksum verification step makes `lfs_read_verify`() slower

than `lfs_read`(), but useful in applications where data correctness is critical. Sensor applications can choose to save or discard corrupt data. The arguments to the functions include the file identifier (`file_id`), the read buffer (`buf`), and the size of the buffer (`size`). The return value indicates the size of the record read. The `lfs_delete`() function takes the file identifier (`file_id`) as input and deletes the oldest record in the file to create space for new records.

`read_from_config`() and `write_to_config`() are used to handle configuration data. Configuration entries are identified using indices into the configuration file. The configuration data file is not a circular buffer, but is stored using the consistent data structure approach for robustness.

5.3 Evaluation

The LoggerFS implementation was evaluated for both reliability and performance. Reliability is measured in terms of the file system's ability to recover from failures while read and write workloads are executing. Performance is characterized in terms of the maximum rate of read and write operations that the file system is able to sustain.

5.3.1 Reliability

The LoggerFS implementation was first validated to ensure its ability to recover from failures. During normal operation, a file system can encounter system failures, such as, power loss due to battery failure, or sub-system failures caused by faulty hardware components. System failures are caused by hardware faults which cause the entire system to go down. Sub-system failures are usually limited to specific hardware components and can be isolated at the software level. We characterize four failure scenarios representative of both types of failures. The four experimental scenarios are enumerated below. To simulate system failures, we introduce power loss during operation (case 1, below). Sub-system failures are simulated by injecting software faults during write operations to flash, FRAM, or both flash and FRAM simultaneously (cases 2 to 4, respectively).

1. **System failure:** During active read and write operations, power was removed from the system, and the system was later checked for consistency after restart.

2. **Flash sub-system failure:** Writes to flash were made to fail at random intervals (by returning failure codes) during write operations, and the system was later checked for consistency.

3. **FRAM sub-system failure:** Writes to FRAM were made to fail at random intervals during write operations to FRAM, and the system was later checked for consistency.

4. **Flash and FRAM sub-system failures:** Writes to either flash or FRAM were made to fail at random intervals during write operations, and the system was later checked for consistency.

Four files, of sizes 1 KB, 2 KB, 4 KB, and 8 KB were created, and a total of 10 trials were run for each of the cases. A 50%-50% mixed read/write workload was issued for case 1, while a 100% write workload was issued for cases 2, 3, and 4.

5.3.1.1 Results

For case 1, we non-deterministically remove power to the MoteStack during 1 minute long mixed read and write workloads. For each test, LoggerFS was able to handle system failures gracefully and re-initialize after power loss. The $lfs\_reinit()$ function was able to verify a consistent version of the file system installed in both FRAM and flash, and then continue normal operation.

For cases 2, 3, and 4, we ran individual trials for a duration of 1 minute. All the tests involved sending write operations to the four files. For case 2, faults were injected at random intervals during write operations to flash. For case 3, faults were injected randomly during writes to FRAM. For case 4, faults were injected at random intervals during writes to both flash and FRAM. After 1 minute, all data was read back and verified. All read and verify operations completed successfully. Further, on the next system reboot, $lfs\_reinit()$ was able to verify a consistent file system installed on FRAM and flash.

5.3.2 Performance

An embedded sensor can experience brief periods of time when it issues 100% writes due to sensing operations. Similarly, there are periods of time when the device issues 100% reads associated with transmission of data to a base station. There are also brief periods of time when mixed workloads might be observed, where a sensing operation is initiated while data transmission is in progress. To evaluate system performance, we first characterized typical I/O workloads and measured system performance at 100% write, 100% read, and mixed read and write workloads. For simplicity of measurement and analysis of mixed workloads, we assumed a 50-50% mix of read and write operations. All read and write operations are 256 bytes.

5.3.2.1 Hardware Benchmarking

We first benchmarked the read and write speeds for the FRAM and flash devices used in the LoggerFS implementation. The results allowed us to better characterize expected read and write performance.

Device	Reads	Writes	Erase and Write
FRAM	0.8 ms	0.6 ms	-NA-
Flash	1.2 ms	1.1 ms	4.3 ms

Table 5.1: Measured Read and Write Speeds

Table 5.1 presents the measured read and write speeds for the FRAM and flash memory devices. We measured the time required to transfer 1 million buffers with 256 bytes of data each, and then computed the average time required to transfer a 256-byte buffer, for both read and write operations. Read operations from flash are 50% slower than reads from FRAM; reading 256 bytes from flash takes 1.2 ms, while it requires 0.8 ms to read from FRAM. Writing 256 bytes of data from flash takes 1.1 ms, which is 83.34% slower than FRAM, which requires 0.6 ms. However, when the time required to erase a block is amortized across write times, flash is about 600% slower, at 4.3 ms. This write speed advantage allows FRAM to handle bursts of high-rate data up to 426.67 KBps, and to sustain write throughputs up to 7 times faster than flash memory, at 59.53 KBps.

5.3.2.2 Write I/O Performance

We measured the write performance of LoggerFS assuming variable degrees of cache availability, including no cache. We considered a 100% write workload and simulated conditions where a percentage of writes cause a cache miss. A 0% cache hit rate represents the case where every new record update causes file eviction(s) from FRAM and a read from flash; whereas a 100% cache hit rate does not cause a file eviction. We create a total of 1,000 files of size 256 bytes each, containing one record each. This allows us to model the amount of data evicted from FRAM and subsequently read in from flash. We know that the size of the evicted file is the same as the size of the file which is being paged in.

Figure 5.9 summarizes the write throughput observed at variable cache hit rates. The x-axis represents the write cache hit rate, and the y-axis represents the throughput achieved. In the absence of the write cache, LoggerFS is able to achieve write throughput of approximately 59.53 KBps. In

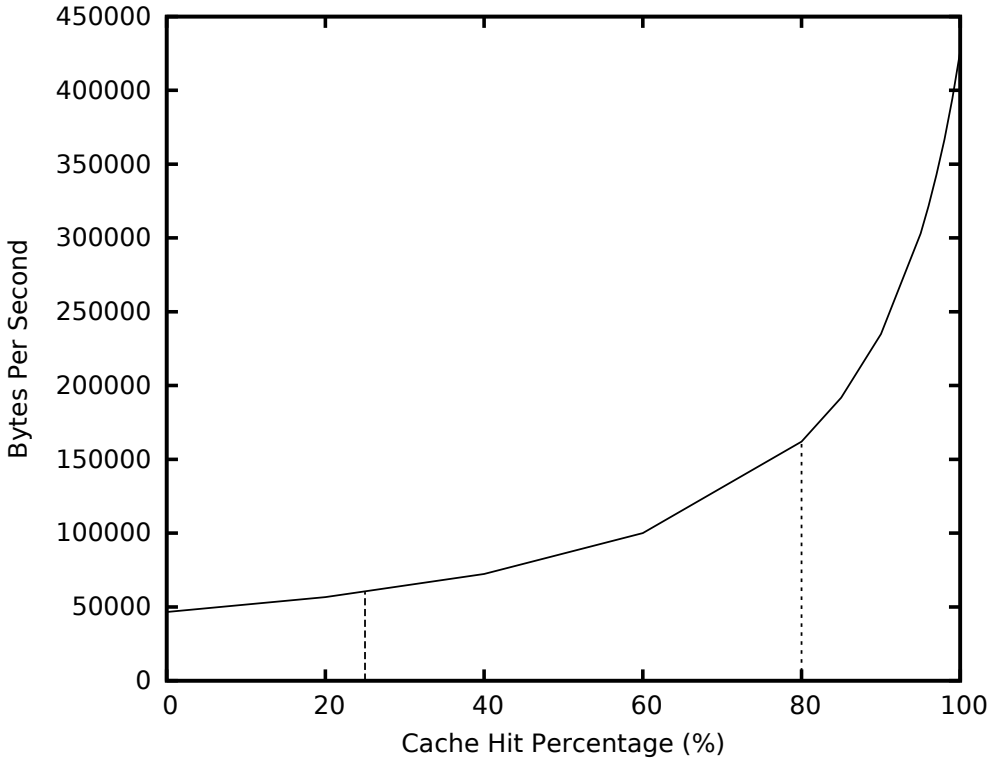


Figure 5.9: Write Throughput with Cache

the presence of a write cache and a 0% hit rate, LoggerFS is only able to achieve write throughput of approximately 46.55 KBps. This can be attributed to the additional overhead associated with file evictions.

With the write cache implementation and a 25% hit rate, LoggerFS achieves write throughput parity with the cache-free implementation. At a cache hit rate of approximately 80%, we observe throughout of 162.02 KBps. Beyond 80%, we observe almost exponential growth in throughput. At 100%, when LoggerFS is able to utilize pure FRAM update speeds, a maximum throughout of 426.67 KBps is observed, which is about 8 times faster than the LoggerFS implementation without cache.

5.3.2.3 Read I/O Performance

We next measured the read performance of LoggerFS with varying cache availability. It is easier to simulate read cache hit rates (rather than writes), as we do not need to account for evictions. A 0% read cache hit rate is created by reading a file which is entirely stored in flash, while a 100% hit rate results in reading all data from FRAM.

Figure 5.10 summarizes the throughput achieved at varying cache hit rates. The x-axis represents the read cache hit rate, and the y-axis represents the throughput achieved. In the absence of a cache, LoggerFS is able to achieve read throughput of about 213.34 KBps. At a 100% cache hit rate, LoggerFS is able to achieve a read throughput of approximately 320 KBps, which is about 50% faster than the throughput achieved without cache. The difference in throughput between 0% and 100% cache hit rates is not large, since flash and FRAM read speeds are similar orders of magnitude.

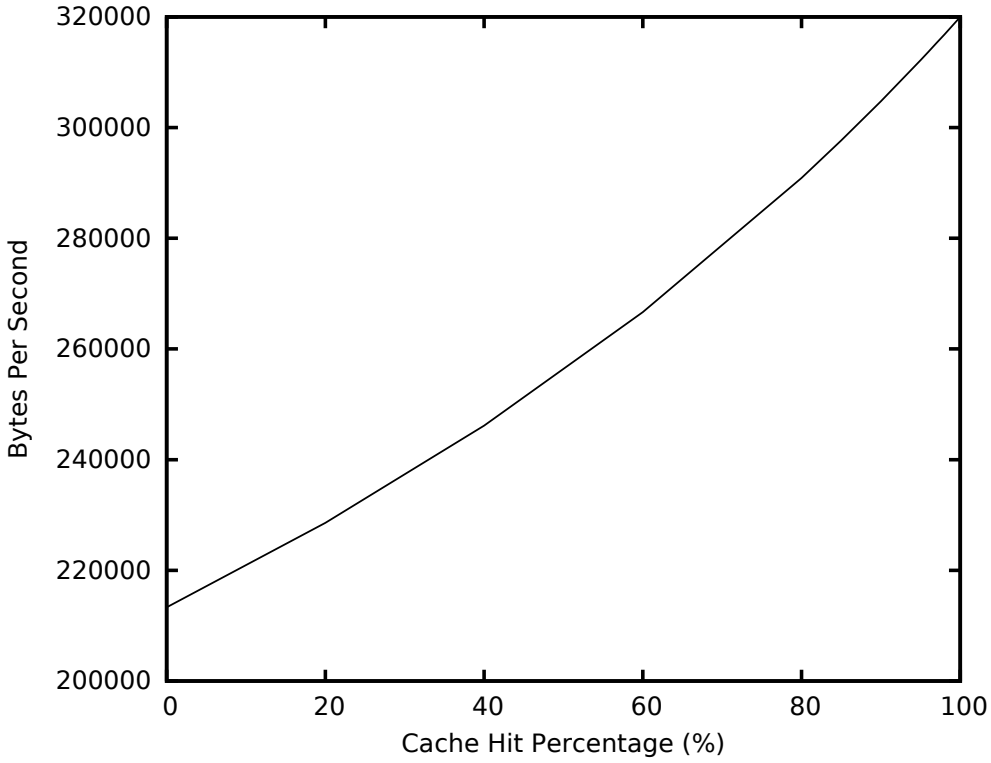


Figure 5.10: Read Throughput with Cache

5.3.2.4 Mixed I/O Performance

Figure 5.11 summarizes the performance for a mixed, 50%/50% read/write workload. The read and write operations in the workload were interleaved. We ran experiments by varying the write cache hit rate from 0% to 100%, in increments of 20%. For each write cache hit rate increment, the read cache hit rates were simultaneously varied from 0% to 100%, in increments of 20%. The x-axis represents the read cache hit rate, and the y-axis represents the throughput achieved. The series represent the various write cache hit rates considered. For example, the solid line charts throughput

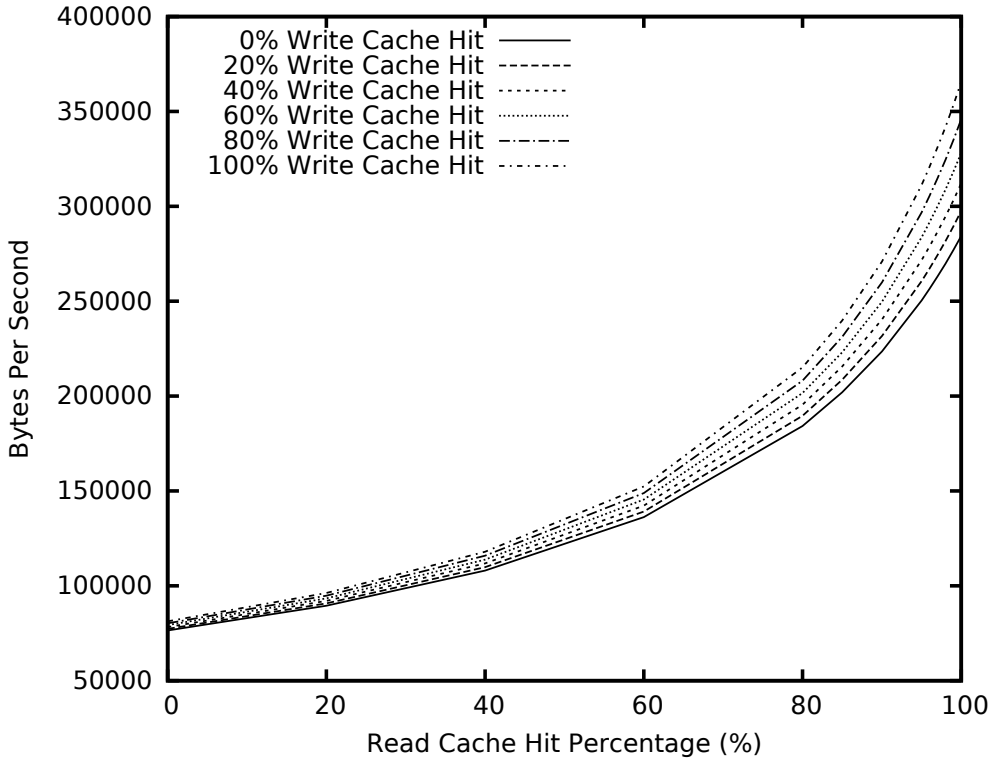


Figure 5.11: 50%/50% Read-Write Throughput

when the write cache hit rate was set at 0%, and the read cache hit rates were changed. The total throughput includes both reads and writes.

Due to the single-threaded nature of LoggerFS, read and write operations are serviced sequentially, and throughput is latency sensitive. We observed that the throughput (at any cache hit rate) can be calculated from the throughput of individual read and write operations from Figures 5.9 and 5.10. The observed throughput at 0% is approximately 76 KBps, while the observed throughput at 100% is approximately 5 times faster, at 365 KBps. The observed data transfer rates track the lower of the read and write throughputs. At lower cache hit percentages, throughput is dominated by write speeds, while at over 80%, the read speeds dominate.

Overall, these results demonstrate that the LoggerFS implementation with the FRAM write-back cache performs about 8 times faster than without the cache for write workloads. For read workloads, the FRAM cache helps boost throughput by 50%. For 50%-50% mixed read and write workloads, LoggerFS with the FRAM write-back cache is able to achieve speeds 5 times faster than the implementation without the cache.

5.4 Summary

In this chapter, we presented the design and implementation of LoggerFS, which provides a fast and reliable data storage solution for WSNs. LoggerFS uses a hybrid memory model comprising RAM, FRAM, and flash memory. The file system is designed to use fixed-size metadata, resulting in a small main memory footprint. Metadata is stored in FRAM for fast reads and writes. Data persistence and reliability are achieved using a consistent data structure update mechanism, ensuring that all updates in the file system are atomic. We presented the design of a FRAM-based, write-back cache, which allows LoggerFS to efficiently handle bursts of high-rate writes, and to support flash wear-leveling. Finally, we demonstrated the fault tolerance capabilities of LoggerFS during various failure scenarios and evaluated the read and write performance of a LoggerFS prototype on an in-situ sensing platform. We demonstrated significant improvements in read and write performance with the FRAM-based write-back cache, over the implementation without cache.

Chapter 6

Related Work

In this chapter, we summarize the most relevant related work. Section 6.1 summarizes work related to our task scheduling approach. Section 6.2 summarizes work related to our network reprogramming approach. Section 6.3 summarizes work related to our fast and reliable data storage solution.

6.1 Task Scheduling

Levis *et al.* present TinyOS[39], one of the most widely-used sensor network operating systems. TinyOS includes a task scheduler that executes non-preemptive tasks *posted* for later execution. TinyOS uses a fixed-length, FIFO scheduler by default. To reduce energy consumption, the scheduler puts the processor to sleep whenever the task queue is empty. Its successor, TinyOS2[38], uses a similar FIFO scheduler; an earliest-deadline-first implementation is also available. Compared to TinyOS, TinyOS2 introduces more overhead when posting and executing a task, but less overhead when the task queue is empty.

Han *et al.* present SOS[25], another event-driven operating system. Software modules communicate using direct calls and message passing via a FIFO scheduler with two levels of priority. High priority messages are reserved for time critical events, such as hardware interrupts.

Dunkels *et al.* present Contiki[21], another event-based operating system with support for event prioritization. A non-preemptive event scheduler schedules asynchronous and synchronous events. *Asynchronous* events are deferred procedure calls enqueued in a FIFO handling queue.

Synchronous events are immediately scheduled at the front of the queue.

Bhatti *et al.* present MANTIS[9], a multi-threaded sensor network operating system. In MANTIS, a fixed thread table maintains all threads, which are executed using round-robin scheduling within priority levels. The scheduler is driven by a timer interrupt, which triggers context switching among threads. MANTIS also allows users to specify the sleep period of threads. The scheduler calculates the earliest wake-up time and uses an idle background thread to put the CPU to sleep when all other threads are blocked.

Chen *et al.* present Enix[12], a cooperative threading solution for sensor networks, which uses *setjump* and *longjump* to implement low overhead context switching. It supports priority-based and round-robin scheduling policies using linear search and bitmap-based thread lookups. Other multi-threaded sensor network operating systems, including LiteOS[10] and RETOS[11], use similar schedulers. In particular, LiteOS supports priority-based and round-robin scheduling policies, and RETOS supports POSIX scheduling, which boosts the priority of a thread when events need to be handled quickly.

While each has its advantages, none of these systems are well matched for AAS scheduling. Event-based schedulers using FIFO mechanisms or priorities are not designed to account for the sleep requirements of AAS systems. Thread-based schedulers are also inefficient in this context. POSIX-like soutions introduce significant overhead, while the use of small epochs in other multi-threaded solutions is energy-inefficient. By contrast, our work focuses on the systematic design and analysis of scheduling solutions suited specifically to AAS systems.

6.2 Network Reprogramming

Levis *et al.* present TinyOS [39], which provides Crossbow Network Programming (XNP) [33, 62] as its network reprogramming implementation. XNP achieves network reprogramming by broadcasting the entire program image to nodes in a single-hop network. Culler et al. present Deluge [27], a reliable data dissemination protocol which also propagates complete binary images. However, both protocols are inefficient; there are often common code segments between versioned images.

Stathopoulos *et al.* present Multihop Over-the-Air Programming (MOAP) [58], which uses a data dissemination protocol called *Ripple* to distribute code to sensor devices. Unlike network *flooding*, Ripple selectively forwards packets to nodes while utilizing a sliding window protocol for

controlling retransmissions. Nodes have the ability to transmit parts of the program code they have already received to new nodes while waiting for retransmission of lost packets.

Levis *et al.* present Maté [37], which deals with network reprogramming by transmitting application-specific code for execution on a virtual machine. While this allows Maté to be significantly faster during reprogramming, the approach is not useful when the virtual machine itself needs to be reprogrammed. Levis *et al.* also present Trickle [40], which uses an epidemic-based data dissemination protocol to avoid flooding the network as in Maté. However, none of these approaches consider incremental code updates for efficient reprogramming.

Jeong describes Fixed Block Comparison (FCB) [31], which divides program images into fixed size blocks and compares the blocks in the corresponding locations in both the old and new program images. FCB then propagates only the blocks of code from the new program image which are different from the previous version. FCB performs only marginally better than XNP when the two program image versions are not aligned with each other.

Jeong *et al.* also present an incremental code update strategy in [32], where they adapt the Rsync algorithm [65] to compute differences between program image versions. The approach again partitions the program image into fixed-size blocks (B bytes), and then uses a checksum pair (checksum, hash value) to represent each block, and stores the pair in a hash table. Next, a sliding window of size B bytes is run on the new program image, and the checksum and hash value for each window are calculated; lookups are performed in the hash table for potential matches. While Rsync is also platform and language independent, there are problems with this approach. The total number of hash computations used in Rsync is proportional to the size of the code image, $O(n)$. Considering that each hash computation requires at least linear time, the time required for all the hash computations and lookups is $O(n^2)$. Second, on a hash match, the approach requires a byte-by-byte scan through the code to avoid false match positives. Finally, the size of the sliding window defines the match granularity in the two image versions. If there are multiple matching image segments of size $(B - b)$ bytes, where $0 < b < B$, this approach would fail to identify the matches.

Panta et al. present Zephyr [48], an incremental reprogramming strategy based on an optimized version of the Rsync algorithm [65], in conjunction with function call indirection. This approach requires application-level code modifications to reduce *function shifts* caused when function bodies are shifted from their original locations between image versions. Next, the optimized Rsync

algorithm is used to compute the differences between the two code images, creating a *delta*. While the traditional Rsync algorithm is able to identify matching blocks, the optimized version computes the *maximal super-block* between the two images in $O(n^2)$ time, where n is the length of the code images. A *super-block* comprises contiguous matching blocks, and a maximal super-block is the largest super-block. While Zephyr is an improvement over the strategy presented in [32], it shares all the problems of the Rsync-based approach. Further, this solution is not platform and programming language independent; it requires knowledge of program structure.

Munawar et al. present Dynamic TinyOS [47], which uses high-level knowledge of application structure to make application updates. This is achieved using extensions to the nesC compiler which convert TinyOS applications and system components into separate binary objects during compilation. Standard data dissemination protocols are then used to update individual objects. This approach also requires knowledge of program code structure, which reduces its applicability to systems developed using other compilers and languages.

Reijers *et al.* describe an efficient code distribution strategy that uses a *diff*-like approach to computing the edit script for encoding the differences between two program images [51]. Their approach makes use of a *suffix tree*, which requires $O(n)$ time and space to build, where n is the length of the original version of the program image. However, their approach needs n traversals of the suffix tree for each position of the image vector, thus requiring $O(n^3)$ time. Additionally, the edit script encoding scheme is complex, requiring a large number of commands and opcodes, and is architecture specific.

In contrast to all the prior approaches, our incremental update strategy uses an adapted version of Hirschberg's Algorithm to compute the differences between program images. Hirschberg's Algorithm has quadratic time and linear space complexity and employs a divide-and-conquer dynamic programming approach to compute a globally optimal subsequence between two strings. We adapt Hirschberg's Algorithm to build the edit map containing the edit script required to transform the code running in the network to a new code image. Since we do not use a block-based approach, our solution is able to identify even small code segments which match between the program images. Further, we present an optimization strategy for encoding the edit map, which significantly reduces the amount of data that needs to be transmitted (and the energy expended) for successful sensor node reprogramming.

6.3 Data Storage

In this section, we survey some of the most well-known file systems for embedded systems, many of which use log-structured design principles [52]. The design of LoggerFS is influenced in part by JFFS [69], designed by Woodhouse *et al.* Prior to JFFS, the traditional design approach was to use a block device interface to interact with flash memory. JFFS was among the first to introduce a log-structured design specifically for flash devices. Versions have been implemented for the 2.x Linux kernels; those support garbage collection and checksum-based fault detection. However, JFFS lacks a fault-tolerance mechanism, including the ability to recover from system crashes. Additionally, although JFFS was designed for use in embedded devices, its implementation does not consider resource-constrained systems.

Mathur *et al.* present Capsule [43], an energy-optimized, NAND flash-based, object storage system for memory-constrained sensor devices. Capsule implements a flash abstraction layer (FAL), which provides a log-structured file system abstracting raw flash access. Capsule supports efficient storage of commonly used objects, such as streams, files, arrays, queues, and lists. Data writes are appended to a fixed-sized write buffer in primary memory, and flushed to flash when full. Reads are always served from flash, not cache. The file system implements a *cleaner* as part of the FAL, responsible for garbage collection of fragmented data in flash. Fault tolerance is provided by checkpointing and rollback of object states. Checkpointing involves taking a snapshot of in-memory state and committing that state to flash. In the event of a node restart, rollback is achieved by restoring the system to the most recent checkpoint. Capsule is implemented using TinyOS [39] and focuses primarily on energy efficiency.

Dai *et al.* present ELF [16], which also implements a log-structured file system for embedded devices. ELF uses a group of pages in a linked-list to represent files in memory. It offers a rich set of features which provide support for random file overwrites and hierarchical directory structures. ELF also allows files to grow in memory after they have been created, handling flash fragmentation via a garbage collector. Interestingly, ELF stores files in NAND flash, and file-system metadata, such as directory structures, in EEPROM to limit the wear on flash. It also supports limited crash recovery for certain special files. Crash recovery metadata is also stored in EEPROM.

Our design differs from ELF and Capsule in that we do not allow files to grow once they have been created. LoggerFS also does not support directory structures and random file overwrites since

embedded sensor file systems are mostly used for logging sensed data. Additionally, the amnesiac nature of LoggerFS (old records are discarded when new data arrives to a full file) allows it to operate without a garbage collector. This makes LoggerFS much faster and lighter-weight, while providing complete file system crash recovery capabilities. Data persistence and crash recovery in LoggerFS is achieved by the use of a consistent data structure update mechanism and a FRAM metadata store. This allows for crash recovery in LoggerFS to be fine-grained, without the need for explicit checkpointing and rollback operations. Finally, the FRAM-based, write-back cache allows LoggerFS to support fast writes, and also serves as an opportunistic read cache.

Doh *et al.* present a file system which uses NVRAM to store file metadata and flash to store file content [18]. They present a model to analyze and predict the amount of NVRAM required for a specific flash size to achieve the optimal usable flash-to-NVRAM ratio for peak performance. For realistic file access workloads, they achieve maximum performance improvements of 600%, and an average improvements of 437% over YAFFS [13], another flash-based file system, successfully demonstrating that flash-based file systems can use NVRAM to accelerate performance. LoggerFS is similar to this file system in that it also employs non-volatile memory (FRAM) to accelerate data access. However, unlike this file system, LoggerFS provides complete crash recovery capabilities.

Tsiftes *et al.* present the Coffee file system [66] for flash-based devices, implemented for the Contiki operating system [21]. Coffee uses a linked-list of page structures to represent files in memory and flash, and uses *micro-logs* to handle file modifications. This allows Coffee to use fixed-size file metadata in RAM. Coffee uses a first-fit algorithm for page allocation in flash, and files are stored as a contiguous list of pages. If files grow larger than their current allocated size, they are moved to a different location in flash. Coffee implements a garbage collector that provides some wear-leveling guarantees. Because of the complex, process-centric implementation of the Coffee file system (requiring significant processing power), LoggerFS outperforms Coffee in terms of read and write performance. Coffee also provides limited crash recovery, making it possible to lose entire file contents during a system failure e.g., due to a loss of power. This lack of data persistence makes Coffee unsuitable for many critical applications.

Gay *et al.* present the Matchbox file system [24] as part of the TinyOS operating system [39], with design goals similar to LoggerFS. Matchbox provides support for data reliability – specifically, the ability to detect corruptions and limit data loss to files being edited during system failure without corrupting file metadata. Matchbox also implements atomic metadata updates, which allows it to

support atomic read and write operations. Memory allocation in flash is done using a free page bitmap. The Matchbox design aligns with many of our design choices: wireless sensor file systems typically do not need support for security, hierarchical file system structures, random file access, or multiple file opens. LoggerFS supports atomic read and write operations via a novel consistent data structure approach, in addition to providing full crash recovery capabilities. Unlike LoggerFS, Matchbox lacks a complete crash recovery solution such that it is able to reinitialize itself after a system failure. Matchbox also requires certain NOR flash capabilities which limit its usability. Additionally, the RAM and ROM footprints for Matchbox are high, increasing linearly with the number and size of files in the system [66]. In comparison, the RAM footprint of LoggerFS is constant for files of any size.

Gal *et al.* present TFFS [23], a transactional file system for resource-constrained embedded devices. TFFS implements an *efficient, pruned, versioned search tree* for file representation in memory; fast search and file access; and atomic operations. Transactions and atomic file operations are stored as part of a log before being committed to memory. For small flash devices, TFFS uses a small amount of primary memory to provide a mapping from real to logical sector numbers. However, for larger flash devices, this memory requirement is much larger, which makes TFFS unsuitable for embedded devices with limited RAM.

Cao *et al.* present a hierarchical file system and an associated shell interface as part of the LiteOS operating system [10]. LiteOS provides a Unix-like file system and operating abstraction for wireless sensor networks, with complete directory structures and user-shell interaction. The unit of storage in the file system is the block. LiteOS stores file metadata (*control blocks*) in EEPROM, and data (*storage blocks*) in flash. An in-memory bit vector provides information about the used blocks in EEPROM and flash. Interestingly, LiteOS also implements a search-by-name feature, which is able to handle string queries. While the LiteOS file system is feature-rich and can be manipulated using shell commands and application development libraries, it is memory-intensive. Further, LiteOS does not guarantee data reliability. The file system may not remain consistent across system failures.

There has been significant work on storage systems for flash-based embedded devices [16, 69, 24, 39, 10, 66, 18, 23]. However, none of these systems are designed to sustain bursty, high data rates, while allowing complete crash recovery and data persistence in the presence of device failures. Moreover, LoggerFS is unique in its lightweight design and its use of FRAM as a metadata store and write-back cache to support fast reads and writes.

Chapter 7

Conclusion

Embedded devices and wireless sensor networks are increasingly pervasive. The design and implementation of the associated applications presents unique challenges to embedded system developers due to the scarcity of computational resources the target hardware devices afford: (1) Most embedded devices operate with finite battery stores. Efficient task scheduling in embedded devices can help extend battery life. (2a) Once these devices have been deployed, it is prohibitively time-consuming and costly to add new functionality or correct even minor defects in installed applications. The ability to efficiently reprogram a wireless sensor network is crucial to propagating code updates without requiring a redeployment. (2b) The absence of associated development, debugging, and reprogramming tools which can be easily modified to suit the developers' needs further complicates the development process. (3) Sensor nodes need persistent data storage. Mission-critical sensor applications need this data to be recoverable after system failures. A file system with integrated reliability mechanisms designed specifically for embedded devices can provide such fault-tolerance, in addition to fast data storage.

7.1 Contribution Summary

In this dissertation, we presented the following contributions.

7.1.1 Contribution 1 — *Task Scheduling*

We presented the design and implementation of four progressively efficient schedulers designed to support *almost-always-sleeping* embedded applications. The scheduler designs are applicable to any modern MCU. We analyzed the scheduler designs and presented closed-form algebraic models for each scheduler implementation, capturing the scheduling overhead of each design as a function of task load and other parameters. Next, these models were used to characterize the comparative performance among the designs. Further, we conducted physical power profiling studies on an in-situ sensing platform and illustrated the power consumption profile associated with each design. The results also provided comparative lifetime benefits. The systematic consideration of this class of schedulers enables embedded system developers to choose the most energy efficient scheduling solution for their application designs, and to extend the battery life of their devices.

7.1.2 Contribution 2 — *Network Reprogramming*

We first presented the design and implementation of a framework for developing and testing incremental code update mechanisms. This unified development framework operates on a set of standardized reprogramming tools and facilitates the implementation and evaluation of incremental code update algorithms for WSN developers. Next, we presented an incremental code update mechanism for efficient wireless sensor network reprogramming, designed using the framework. Our network reprogramming approach adapts Hirschberg's Algorithm to compute the differences between two program images. The differences are stored as an edit script, and a heuristic-based optimization strategy is applied to reduce its size. This optimized edit script is then transmitted to sensor nodes using a standard data dissemination protocol. The sensor nodes construct the new program image after decoding the edit script. Our approach significantly reduces the data size of image transmissions over full image transmissions, and also improves on prior work in incremental reprogramming. Our approach is platform, programming language, and program code structure agnostic. The ability to efficiently reprogram any wireless sensor network will reduce the time required to deploy these networks and modify installed applications, and lead to significant energy savings.

7.1.3 Contribution 3 — *Data Storage*

We presented the design and implementation of LoggerFS, a fast and lightweight file system for wireless sensor systems, which is reliable across device failures, safeguarding the data recorded by these devices. With the use of a hybrid memory model comprising RAM, FRAM, and flash devices, LoggerFS allows sensed data to be sampled and stored quickly and batched for later transmission. The file system has a small main memory footprint, due to its use of fixed-sized metadata. File metadata is stored in FRAM for fast retrieval during read and write operations. A FRAM-based, write-back cache allows bursts of high-rate writes to be buffered, with support for flash wear-leveling. The write-back cache also helps some read workloads by opportunistically serving data from the cache. Data persistence and reliability are achieved using a consistent data structure update mechanism which ensures that all updates in the file system are atomic. LoggerFS is able to provide reliable data storage, while also demonstrating significant improvements in read and write speeds (between 50% and 800%) when using the FRAM write-back cache over an implementation without the cache. This allows LoggerFS to be used in mission-critical applications, where fast and persistent data storage is important.

7.2 Expected Impact

We believe that these contributions can have a significant impact on the design, development, and deployment of applications for wireless sensor networks. First, the design and analysis of the *almost-always-sleeping* schedulers enables system developers to choose the most power-efficient scheduling solution for their applications, thus extending battery life. Second, the network reprogramming solution shortens deployment life cycles by providing the ability to efficiently reprogram a wireless sensor network and modify installed applications with relative ease, while also extending battery life. Further, the framework for developing and testing incremental code update mechanisms allows for rapid prototyping and development of new and improved network reprogramming solutions. Finally, the data storage solution allows for fast, persistent, and reliable storage of data, while using a small memory footprint, and supporting bursts of high data-rate writes. This enables system developers to use these embedded devices for mission-critical applications.

Bibliography

[1] J. Aas. Understanding the linux 2.6.8.1 cpu scheduler. *Silicon Graphics International, 2005. http://josh.trancesoftware.com/linux/linux_cpu_scheduler.pdf.*

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38:393–422, 2002.

[3] Atmel. 4-megabit spi serial flash memory. http://media.digikey.com/pdf/Data Sheets/Atmel PDFs/AT25DF041A.pdf.

[4] Atmel. Avr068: Stk500 communication protocol. www.atmel.com/images/doc2591.pdf.

[5] Atmel. Avr069: Avrisp mkii communication protocol. www.atmel.com/images/doc8015.pdf.

[6] Atmel. Avr910: In-system programming. www.atmel.com/images/doc0943.pdf.

[7] AVR. Libc. http://www.nongnu.org/avr-libc/.

[8] AVR. Atmega datasheet. www.atmel.com/dyn/resources/prod_documents/doc2593.pdf, 2010.

[9] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, August 2005.

[10] Qing Cao, Tarek Abdelzaher, John Stankovic, and Tian He. The liteos operating system: Towards unix-like abstractions for wireless sensor networks. In *Proceedings of the 7th international conference on Information processing in sensor networks*, IPSN '08, pages 233–244, Washington, DC, USA, 2008. IEEE Computer Society.

[11] Hojung Cha, Sukwon Choi, Inuk Jung, Hyoseung Kim, Hyojeong Shin, Jaehyun Yoo, and Chanmin Yoon. Retos: resilient, expandable, and threaded operating system for wireless sensor networks. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 148–157, New York, NY, USA, 2007. ACM.

[12] Yu-Ting Chen, Ting-Chou Chien, and Pai H. Chou. Enix: a lightweight dynamic operating system for tightly constrained wireless sensor platforms. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 183–196, New York, NY, USA, 2010. ACM.

[13] Aleph One Company. Yaffs (yet another flash file system). http://www.aleph1.co.uk/yaffs/yaffs.html.

[14] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reily, 2005.

[15] Rone Ilídio da Silva, Virgil Del Duca Almeida, André Marques Poersch, and José Marcos Silva Nogueira. Spatial query processing in wireless sensor network for disaster management. In *Proceedings of the 2nd IFIP conference on Wireless days*, WD'09, pages 194–198, Piscataway, NJ, USA, 2009. IEEE Press.

[16] Hui Dai, Michael Neufeld, and Richard Han. Elf: an efficient log-structured flash file system for micro sensor nodes. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 176–187, New York, NY, USA, 2004. ACM.

[17] Brian S. Dean. Avrdude - avr downloader/uploader. www.nongnu.org/avrdude/.

[18] In Hwan Doh, Jongmoo Choi, Donghee Lee, and Sam H. Noh. Exploiting non-volatile ram to enhance flash file system performance. In *Proceedings of the 7th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '07, pages 164–173, New York, NY, USA, 2007. ACM.

[19] Cypress Semiconductor Doug Mitchell. nvsrams eclipse battery-backed memory. http://www.cypress.com/?docID=9279.

[20] Adam Dunkels. *Programming Memory-Constrained Networked Embedded Systems*. PhD thesis, SICS, 2007.

[21] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, LCN '04, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[22] G. W. Eidson, S. T. Esswein, J. B. Gemmill, Jason O. Hallstrom, T. R. Howard, J. K. Lawrence, Christopher J. Post, C. B. Sawyer, Kuang-C. Wang, and D. L. White. The south carolina digital watershed: End-to-end support for real-time management of water resources. *IJDSN*, 2010, 2010.

[23] Eran Gal and Sivan Toledo. A transactional flash file system for microcontrollers, 2005.

[24] David Gay. Matchbox: A simple filing system for motes. http://www.docs.tinyos.net/tinyos-1.x/doc/matchbox.pdf.

[25] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, MobiSys '05, pages 163–176, New York, NY, USA, 2005. ACM.

[26] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, June 1975.

[27] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *In Proceedings of the 2nd international*, pages 81–94. ACM Press, 2004.

[28] Texas Instruments. Msp430 ultra-low-power microcontrollers. http://www.ti.com/lit/sg/slab 34w/slab034w.pdf.

[29] Intel. Intel hexadecimal object file format specification. www.microsym.com/editor/assets/intel hex.pdf, 1988.

[30] Digi International. Xbee/xbee-pro oem rf modules - 802.15.4 protocol. ftp://ftp1.digi.com/sup port/documentation/90000982_A.pdf, 2008.

[31] Jaein Jeong. Node-level representation and system support for network programming, 2003.

[32] Jaein Jeong and David Culler. Incremental network programming for wireless sensors. In *IEEE Sensor and Ad Hoc Communications and Networks (SECON*, pages 25–33, 2004.

[33] Jaein Jeong, Sukun Kim, and Alan Broad. Network reprogramming. TinyOS document, http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/NetworkReprogramming.pdf.

[34] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steve Glaser, and Martin Turon. Wireless sensor networks for structural health monitoring. In *Proceedings of the 4th international conference on Embedded networked sensor systems*, SenSys '06, pages 427–428, New York, NY, USA, 2006. ACM.

[35] Claudio Lanconelli. Ponyprog serial device programmer. http://www.lancos.com/prog.html.

[36] Hyung Gyu Lee and Naehyuck Chang. Low-energy heterogeneous non-volatile memory systems for mobile systems. *Journal of Low Power Electronics*, 1:52–62, 2005.

[37] Philip Levis and David Culler. Maté: a tiny virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(5):85–95, October 2002.

[38] Philip Levis, David Gay, Vlado H, Jan hinrich Hauer, Ben Greenstein, Martin Turon, Jonathan Hui, Kevin Klues, Cory Sharp, Robert Szewczyk, Joe Polastre, Philip Buonadonna, Lama Nachman, Gilman Tolle, David Culler, Adam Wolisz, Technische Universitt Berlin, Crossbow Inc, and Arched Rock Corpration. T2: A second generation os for embedded sensor networks. Technical report, 2005.

[39] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. Tinyos: An operating system for sensor networks. In *in Ambient Intelligence*. Springer Verlag, 2004.

[40] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *In Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI*, pages 15–28, 2004.

[41] Konrad Lorincz, Bor-rong Chen, Geoffrey Werner Challen, Atanu Roy Chowdhury, Shyamal Patel, Paolo Bonato, and Matt Welsh. Mercury: a wearable sensor network platform for high-fidelity motion analysis. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 183–196, New York, NY, USA, 2009. ACM.

[42] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, WSNA '02, pages 88–97, New York, NY, USA, 2002. ACM.

[43] Gaurav Mathur, Peter Desnoyers, Deepak Ganesan, and Prashant Shenoy. Capsule: An energy-optimized object storage system for memory-constrained sensor devices. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 195–208, New York, NY, USA, 2006. ACM.

[44] Biswajit Mazumder and Jason O. Hallstrom. Sfc: a simple flow control protocol for enabling reliable embedded network systems reprogramming. In *Proceedings of the 50th Annual Southeast Regional Conference*, ACM-SE '12, pages 321–326, New York, NY, USA, 2012. ACM.

[45] Biswajit Mazumder and Jason O. Hallstrom. An efficient code update solution for wireless sensor network reprogramming. In *Proceedings of the eleventh ACM international conference on Embedded software*, EMSOFT '13, New York, NY, USA, 2013. ACM.

[46] Prashanth Mohan, Venkata N. Padmanabhan, and Ramachandran Ramjee. Nericell: rich monitoring of road and traffic conditions using mobile smartphones. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 323–336, New York, NY, USA, 2008. ACM.

[47] Waqaas Munawar, Muhammad Hamad Alizai, Olaf Landsiedel, and Klaus Wehrle. Dynamic tinyos: Modular and transparent incremental code-updates for sensor networks. In *ICC'10*, pages 1–6, 2010.

[48] Rajesh Krishna Panta, Saurabh Bagchi, and Samuel P. Midkiff. Zephyr: efficient incremental reprogramming of sensor nodes using function call indirections and difference computation. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 32–32, Berkeley, CA, USA, 2009. USENIX Association.

[49] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. Cflru: a replacement algorithm for flash memory. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, pages 234–241, New York, NY, USA, 2006. ACM.

[50] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: enabling ultra-low power wireless research. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press.

[51] Niels Reijers and Koen Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, WSNA '03, pages 60–67, New York, NY, USA, 2003. ACM.

[52] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10:1–15, 1992.

[53] Cypress Semiconductor. Cy14b104n datasheet. http://www.cypress.com/?rID=39015.

[54] Cypress Semiconductor. Fm22l16 4mbit asynchronous f-ram memory. http://www.cypress.com/?docID=42532.

[55] Cypress Semiconductor. Fm25v05 512-kbit serial f-ram. http://www.cypress.com/?docID=47930.

[56] Victor Shnayder, Mark Hempstead, Bor-rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 188–200, New York, NY, USA, 2004. ACM.

[57] Livio B Soares, Orran Y Krieger, and Dilma Da Silva. Meta-data snapshotting: A simple mechanism for file system consistency, 2003.

[58] Thanos Stathopoulos, John Heidemann, and Deborah Estrin. A remote code update mechanism for wireless sensor networks. Technical report, 2003.

[59] Crossbow Technologies. Iris datasheet. http://bullseye.xbow.com:81/Products/Product_pdf_files/Wireless_pdf/IRIS_Datasheet.pdf.

[60] Crossbow Technologies. Mica2 datasheet. http://bullseye.xbow.com:81/Products/Product_pdf_files/Wireless_pdf/MICA2_Datasheet.pdf.

[61] Crossbow Technologies. Micaz datasheet. http://bullseye.xbow.com:81/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf.

[62] Crossbow Technology. Mote in network programming user reference. TinyOS document, http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/Xnp.pdf.

[63] Andreas Terzis, Razvan Musaloiu-E., Joshua Cogan, Katalin Szlavecz, Alexander Szalay, Jim Gray, Stuart Ozer, Chieh-Jan Mike Liang, Jayant Gupchup, and Randal Burns. Wireless sensor networks for soil science. *Int. J. Sen. Netw.*, 7(1/2):53–70, February 2010.

[64] Gilman Tolle, Joseph Polastre, Robert Szewczyk, David Culler, Neil Turner, Kevin Tu, Stephen Burgess, Todd Dawson, Phil Buonadonna, David Gay, and Wei Hong. A macroscope in the redwoods. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, SenSys '05, pages 51–63, New York, NY, USA, 2005. ACM.

[65] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, 1999.

[66] Nicolas Tsiftes, Adam Dunkels, Zhitao He, and Thiemo Voigt. Enabling large-scale storage in sensor networks with the coffee file system. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, IPSN '09, pages 349–360, Washington, DC, USA, 2009. IEEE Computer Society.

[67] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 381–396, Berkeley, CA, USA, 2006. USENIX Association.

[68] Geoffrey Werner-Allen, Stephen Dawson-Haggerty, and Matt Welsh. Lance: optimizing high-resolution signal collection in wireless sensor networks. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 169–182, New York, NY, USA, 2008. ACM.

[69] David Woodhouse. JFFS : The Journalling Flash File System, 2001.

[70] Ning Xu, Sumit Rangwala, Krishna Kant Chintalapudi, Deepak Ganesan, Alan Broad, Ramesh Govindan, and Deborah Estrin. A wireless sensor network for structural monitoring. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 13–24, New York, NY, USA, 2004. ACM.

[71] Tom Yeh, Haru Yamamoto, and Thanos Stathopolous. Over-the-air reprogramming of wireless sensor nodes. In *UCLA EE202A Project Report*, 2003.