TEARING DOWN SILOS IN IOT SYSTEMS

by

Neil Klingensmith

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Electrical Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2019

Date of final oral examination: 06/25/2019

The dissertation is approved by the following members of the Final Oral Committee:

Suman Banerjee, Professor, Computer Sciences
Giri Venkataramanan, Professor, Electrical and Computer Engineering
Nancy Wong, Professor, Consumer Science
Younghyun Kim, Assistant Professor, Electrical and Computer Engineering

© Copyright by Neil Klingensmith 2019
All Rights Reserved

To my wife, Kelly Klingensmith,
and my parents, Wally and Sandy Klingensmith.

ACKNOWLEDGMENTS

Like many people who pass through the halls of academia, my PhD began with a desire to remain out of the workforce. Now, seven years in, it has come to define my career, my family, and many other aspects of life.

I would like the thank my advisor Professor Suman Banerjee for supporting my academic inquiry through my PhD and for allowing me to pursue my own direction. Suman and I met by accident in the Summer of 2009 while I was an undergraduate. As a graduate student, Suman gently guided me in the direction he thought was best while allowing me to make my own mistakes. For me, his advising style was effective: I bumped into a lot of walls and fell down a lot of holes in grad school, but I learned the landscape of systems research.

I would also like to thank Professor Giri Venkataramanan for his mentorship during my PhD. As a junior graduate student, Giri helped me to get my thesis started. He contributed to several sustainability projects that came to define the direction of my dissertation. Giri was a great collaborator during several research projects, and I appreciate his input on research problems and career advice. Giri has a unique perspective and a well balanced view of the world which is valuable in academia.

I would like to thank Professor Nancy Wong for her collaboration on research projects and for serving on my graduate committee. Nancy's contributions to the sustainability projects in the Wisely lab—which are the central topic of this dissertation—were especially helpful because she provided a perspective from outside of computer science. The approach we took in that line of work was guided a lot by her input.

I would like to thank Professor Younghyun Kim, who I got to know toward the end of my time as a graduate student, for his collaboration on research projects, for serving on my graduate committee, and for sharing his insights into the world of academia. Younghyun is a bit farther ahead in his career than I am, and he has provided some very useful advice and feedback about being an early career academic. Younghyun is a hard worker, and I appreciate that his door was always open.

I would like to thank my wife Kelly for being supportive throughout my PhD. We started dating around the same time that I started graduate school, and she put up with being married to a student for a lot of years. A lot of people would have probably been annoyed with being around a guy who many times didn't know what day of the week it was. But not Kelly. If hanging around a student all the time bothered her, she didn't show it. Together we've bought a couple of houses, had a couple of kids, and done other things that I did not think were possible for a student.

I could not have finished my PhD without guidance and support from my parents Wally and Sandy. They set a good example and applied an appropriate amount of pressure. They served as a sounding board for some important decisions about school, career, and life in general.

I would also like to thank my colleague Zach LaVallee for his collaboration on several research projects. Zach is one of the most optimistic people I have ever known, and I consider myself fortunate to have had the opportunity to work with him. He often sees opportunities in situations that most people would consider obstacles. I would also like to thank Vivek Shrivastava and Vlad Brik, two of my mentors as a young graduate student. They both taught me what to do in grad school and how to be a successful student.

I would like to thank my colleague Ashray Manur for being a good friend throughout graduate school. Ashray and I loosely collaborated on several sustainability projects early on in our time at UW Madison, and he has become an excellent academic and a great guy to bounce ideas off of.

CONTENTS

Contents iv

List of Tables vi

List of Figures viii

1 Introduction 1
- *1.1 Focus of this Thesis* 3
- *1.2 Contributions* 12
- *1.3 Organization* 16

2 Emonix: An IoT System to Monitor Energy Consumption by Plug Loads 18
- *2.1 The EMONIX Energy Measurement and Monitoring Infrastructure* 23
- *2.2 Implementation Challenges* 28
- *2.3 Use Cases in Residence Halls* 30
- *2.4 Summary of Emonix* 33

3 Hot, Cold, and In Between: An IoT Platform to Control HVAC Systems for Comfort and Efficiency 34
- *3.1 Understanding Existing Heating and Cooling Systems* 38
- *3.2 Augmenting the Central HVAC System with Localized Control* 42
- *3.3 Analysis* 50
- *3.4 Summary of Hot, Cold, and In Between* 53

4 AWESOME & SPOCK: An IoT System to Improve the Efficiency of Water Treatment Systems 55
- *4.1 Background* 60
- *4.2 Description of AWESOME* 62
- *4.3 SPOCK Forecasting Algorithm* 69
- *4.4 Evaluation* 77
- *4.5 Summary of Awesome and SPOCK* 87

5 Hermes: A Hypervisor for Mobile and IoT Devices 88

5.1 Background 91

5.2 Architecture 95

5.3 Evaluation 106

5.4 Summary of Hermes 119

6 Related Work 121

6.1 Energy Monitoring 121

6.2 Heating and Cooling Monitoring and Optimization 123

6.3 Building Water Treatment System Optimization 123

6.4 Microcontroller Operating Systems and Hypervisors 125

7 Conclusions 127

7.1 Lessons Learned 127

7.2 Future Work 127

7.3 Relevance to Trends in IoT 130

7.4 Concluding Remarks 131

Bibliography 133

LIST OF TABLES

1.1 Organization of this dissertation and relation to the author's prior work. 17

2.2 Features of Commercial Power Meters. Emonix provides all features necessary in our deployment. 20
2.1 Estimated cost of commercial energy sensors for a panel with 30 breakers. The cost for Emonix reflects only price and assembly. However, our devices are purchased at low volume, so our component cost are higher than they would be for large volumes. These figures include estimated costs for all components of the system, including sensors, current transformers, power supplies, etc. For Emonix, these figures only reflect hardware costs, while commercial products necessarily include costs of doing business such as technical support, engineering time, etc. 21
2.3 Features supported by Emonix's API. 27

3.1 Airflow measurements taken at each home in our study. There was roughly a 50% decrease in conditioned airflow to the second floor. 39
3.3 Energy savings that resulted from the use of localized control in Home C. 50
3.2 Set points of the programmable thermostat in Home C before and after adding localized control. 50

4.1 Features used by our adaptive control algorithm to determine whether the softener system needs to be regenerated. 66
4.2 Ranges of the cost function $C_{\mathtt{diff}}$, and the corresponding sensor readings that cause them. 75
4.3 Comparison of salt consumption in pilot buildings 81
4.4 Improvements of the oracle algorithm and SPOCK over the reserve capacity. 87

5.1 List of privileged instructions that do not cause privilege violations when executed in unprivileged mode. 103
5.2 Entropy of the distributions of latency measurements (distributions shown in Figure 5.4). Low values of entropy are more deterministic. The bare metal guest running in Hermes has much more predictable latency than tasks in FreeRTOS. Under Hermes, latency is still highly deterministic under high I/O load. 107
5.3 Performance comparison of Hermes and FreeRTOS in the video app. Low frame loss rate and jitter is better. 111
5.4 Comparison of ping round trip times in Hermes for three Ethernet driver implementations on the ARM device. 115
5.5 Comparison of SD card write throughput for three driver implementations on the ARM device. 115

LIST OF FIGURES

1.1 High-level goals of this thesis in relation to the individual projects that address them. 4
1.2 Overview of the building automation components presented in this thesis within a home. 8
1.3 Cloud services model for implementing inter-device APIs. 8

2.1 Custom energy sensor board. Monitors up to 16 current transformers and has three distinct forms of communication. 23
2.2 Connection of energy sensor board to monitoring device in breaker panel. Each branch circuit is monitored by a single current transformer, which is an analog device that can sense the current flowing through a conductor. The energy sensor board digitizes the analog signal generated by the current transformer and transmits the data to a gateway. 23
2.3 Multiple energy sensor boards connected to a backplane. 26
2.4 Total energy consumption per resident for the duration of our deployment for each month. 31
2.5 Average energy consumption as a function of hour of the day. Different lines correspond to different randomly chosen residents. 31
2.6 Hypothetical energy bill for several residents. 32

3.1 Floorplan of Home A in our study, showing the placement of temperature sensors throughout the building. Each sensor connects to the Internet through an existing WiFi access point. A remote server collects the data and controls local heating and cooling appliances through network-connected smart power strips. 36
3.2 Without correction, even rooms close to the furnace or air conditioner can experience wildly fluctuating temperatures. In the den of Home C, high-temperature air from the furnace quickly heats the room above its setpoint. When the furnace turns off, heat leaks away through the three exterior walls. 41

3.3 Photograph of temperature sensors used to monitor the environmental conditions in each room. 42
3.4 Photographs of our control equipment. (a) A register booster fan controlled by Emonix smart power strip. (b) An Emonix power strip controls a space heater that provides additional heat in rooms that are too cold. 42
3.5 The internals of an Emonix smart power strip include a relay to switch power to the plugs it serves and an analog adapter board to sense the power consumption of the connected appliances. The main CPU sits beneath the analog adapter board along with its WiFi interface that directly connects to the home's network. This allows energy measurements to be relayed to an off-site controller. 46
3.6 Deviations from the setpoint temperature in the guest bedroom of Home C. Before installing register booster fans, the room was on average 2.5°C below the thermostat's setpoint. Booster fans decreased the average deviation to 1.1°C below setpoint. 47
3.7 Histogram of temperature deviations from the setpoint in the guest bedroom of Home C before and after the installation of register booster fans. The use of register booster fans decreases the magnitude of the average temperature error by 1.4°C. 48
3.8 Observed room temperatures for two rooms on an average day in December 2013. The x-axis is the time of day, and the y-axis is the measured temperature. The master bedroom is located on the second floor and receives less air flow from the furnace, resulting in dramatic temperature reductions. 51
3.9 Observed room temperatures for the master bedroom and dining room after adding localized control to master bedroom. The x-axis is the time of day, and the y-axis is the measured temperature. A space heater is used to increase the master bedroom temperature from 6PM-10PM and from 4:30AM-6:30AM. For the rest of the day, the temperature in the master bedroom is allowed to drift away from its setpoint. 51

4.1 Dataflow diagram for the AWESOME system. Components with solid lines are part of AWESOME/SPOCK, and components with dashed lines are part of the existing softener system. 56
4.2 Reserve capacity (x-axis) vs. regeneration volume (y-axis) for flow data collected in a senior living facility over the course of 225 days in 2015-2016. For the same flow data, we simulated reserve capacity-based regenerations, using the same theoretical water softener tank capacity of 36k gallons. We varied the reserve capacity from 2k to 26k gallons to determine what the optimal setting should be. To ensure that we do not overrun the theoretical capacity of the softener, we must set the reserve capacity to 18k gallons, which wastes 6k gallons per cycle on average—more than 17% of the theoretical capacity. . 63
4.3 A diagrammatic overview of the way a water softener functions, explained in section 4.1. 65
4.4 Photograph of an AWESOME board. 65
4.5 A demonstration of how the features used by the SVM differ for soft water, truly hard water, and false spikes (noise). 66
4.6 Example trace of some raw hardness data gathered from the Calcium ISE. At night, the flow through the water softener drops, creating anomalous spikes that could be confused for hard water output. On 9-12, the sensor output drifts over the course of several hours, even though the water hardness remains constant. 66
4.7 Top: Water flow and hardness measurements (y-axis) as a function of time (x-axis). Bottom: our cost function (y-axis) as a function of time (x-axis). Large positive values of the cost function indicate high flow & low hardness. 76
4.8 A histogram of flow rate measurements taken in a residential building over several months in 2015. The distribution of water flow rate changes depending on hour of the day, so the flow rate function is not stationary. 76

4.9 The hardness at which each algorithm detected filtration medium depletion. We show the detection level for each of five different training levels. Our goal is to detect medium depletion at the lowest possible hardness level while minimizing the number of false positive identifications 77

4.10 False positive rates for the learning algorithms evaluated. Each algorithm was tested with five different training thresholds. In general, lower training thresholds result in more false positives. . 77

4.11 Receiver operating curve for the learning algorithms we analyzed. Algorithms that perform well will have ROCs that are close to the upper left corner of the graph. In that region, the algorithm generates a low fraction of false positives and a large fraction of true positives. 78

4.12 Salt usage in Chadbourne Hall over a two-year timespan. Salt consumption declined dramatically following the installation of AWESOME (solid blue line). This data is based on salt deliveries made by Kreger Salt Sales, the supplier of softener salt for Chadbourne Hall. 80

4.13 Amount of salt used per gallon of water treated in each of the three buildings where we deployed AWESOME. 80

4.14 Relative prediction error of cumulative water flow (y-axis) as a function of the forecast horizon (x-axis). The predictions evaluated here are based on 14 days of historical data. 83

4.15 Relative prediction error of instantaneous water flow (y-axis) as a function of the forecast horizon (x-axis). The predictions evaluated here are based on 14 days of historical data. 83

4.16 Relative prediction error of water hardness (y-axis) as a function of the forecast horizon (x-axis). The predictions evaluated here are based on 14 days of historical data. 84

4.17 Receiver operating characteristic for SPOCK's hardness forecaster. 86

4.18 Comparison of tank capacity using (i) oracle scheduler, (ii) SPOCK autoregressive scheduler, and (iii) reserve capacity. SPOCK performs nearly as well as the oracle predictor, which knows exactly what the sensor readings will be for all future readings. 86

5.1 Timeline of (a) high-priority ISR followed by user-mode I/O processing and (b) low-priority ISR co-occurring with a high-priority ISR, delaying high-priority user mode processing. 90
5.2 Sequence of events required in an RTOS to disable low-priority interrupts in a high-priority ISR. Interrupts would ostensibly be re-enabled in user mode after the I/O event has finished processing. However, the RTOS could transfer control to a different task, which could cause a deadlock. 93
5.3 Architecture of the Hermes Hypervisor. The main component of Hermes, its monolithic exception handler, intercepts all exceptions before dispatching them to the guests. 93
5.4 ISR-userspace latency histograms. Latency is measured as the number of cycles elapsed between executing the serial port receive ISR and beginning of userspace processing. 106
5.5 Number of frames dropped by FreeRTOS per second as a function of time while camera was running in the presence of a ping flood. 113
5.6 Histograms of speed estimates by our GPS app. 113
5.7 Histograms of inter-frame spacing for the video app. Standard deviation of these histograms are *jitter* in Table 5.3. 113

TEARING DOWN SILOS IN IOT SYSTEMS

Neil Klingensmith

In this dissertation we discuss techniques for deconstructing the silos that control IoT and mobile services and house their data. In context of the Internet of Things, silos are self-contained vertical data paths that permit information and commands to flow in a protected pipe without accessibility from external apps or services. Cloud-based services are generally viewed as being on the top of the silo and end devices on bottom. Silos may also include special-purpose intermediate middleboxes such as data aggregators or gateways that do not implement standardized communication technology and work only with a proprietary set of IoT devices. Google's Nest, for example, makes tightly integrated thermostats, doorbells, cameras, and other personal IoT devices that work seamlessly together and cooperate with a common cloud service. The Nest mobile app and hub serve as the main points of user interaction, and they can be used seamlessly with all of Nest's products.

Silos threaten the usability of the Internet of Things in many ways. When service providers wall off their systems, they eliminate the possibility of cooperating with other IoT systems. Silos can also drive up costs for users by requiring them to buy and maintain special-purpose middleboxes.

This dissertation presents practical systems that reduce the need for data and service siloing. It includes a software package for dropping these techniques into existing platforms. Virtualization allows the implementation of device agnostic services to interact with sensors and provide standard APIs on many heterogeneous IoT devices. Although virtualization enables deployment of homogeneous interfaces on heterogeneous devices, it incurs a performance penalty in return for flexibility. The virtualization techniques in this dissertation also pose a deployability challenge: installing custom software on propriety hardware is difficult even for experienced users. We discuss ways in which our techniques could be cleanly integrated with existing platforms to improve performance and simplify software and API customization.

1 INTRODUCTION

One of the most exciting promises of the Internet of Things (IoT) is the prospect of integrating distributed sensing and cloud services to control the built environment with coordinated actions. Learning context from seemingly unrelated input sources is central to this goal. Synergy of services, in this vision, is different from the combination of its components not just in magnitude but in kind—sensors and controllers cooperate at scale and in aggregate to learn users' behavior patterns and anticipate their needs. Systems can take hints about the user's future needs, learned from volumes of historical sensing data, and they can coordinate accordingly. For instance, a smart home might turn on the air conditioner to cool down a few rooms when its user's wearable sensors indicate that he is coming home from a long jog.

But, despite over two decades of research in the area, this vision has not yet materialized. One of the biggest challenges in mobile and IoT computing is data and service siloing—the tendency of individual services and components not to integrate well with one another. Nest thermostats, for instance, do not seamlessly interface with Apple HomeKit. Lack of integration makes service coordination among devices virtually impossible. By isolating components, we are significantly underutilizing the capacity of our deployed devices.

Users, for their part, are comfortable with silos. Refillable mops, brooms, coffee machines, and air fresheners—classical products unrelated to IoT—often work only with the original manufacturer's consumable refills. Even matching furniture and appliances can be thought of as a form of product siloing. But only with mobile and IoT devices does siloing reduce the product's usability.

It doesn't have to be this way. Inter-service cooperation could be established and maintained using secure device-level APIs that permit data and configuration sharing among services. Indeed, some services already provide APIs, but they are not widely used by other potentially synergistic platforms.

Some researchers have attempted to ameliorate the situation by building "data marketplaces" that allow platforms to share data in a centralized repository. But support for marketplaces has not been implemented by any popular service

provider. Possible reasons for this are diverse: building support for APIs and marketplaces is expensive, and most users don't view siloing as a problem. Perhaps most importantly, businesses view siloing as a way of protecting market share. And broad-based data collection—increasingly viewed as corporate surveillance—can compromise user privacy unless service providers are careful and intentional about anonymizing the data they collect. Regardless of the perception of consumers or the business strategy of manufacturers, siloing stands in the way of a ubiquitous Internet of Things.

Definitions In this dissertation, the term IoT *devices*, refers to physical pieces of hardware that connect to the Internet. Devices are the things in the Internet of Things. The term *services* refers to the cloud or edge based backends that store data, implement algorithms, or facilitate user interaction. At present, devices and services are normally connected with proprietary APIs, and some manufacturers also support more limited externally facing public APIs that allow authenticated third parties to interact with their systems. It is these proprietary APIs and their attendant lack of interoperability that is the focus of this dissertation.

As researchers work to smooth the integration of IoT silos, the field of devices continues to expand. For the past several years, enthusiastic and ambitious projections have been made for the rapid growth of the Internet of Things ecosystem. Intel, for instance, has predicted that by the year 2020, the number of connected IoT devices will grow to around 200 billion worldwide, which is more than 20 devices for every person [33]. However, in domestic and personal sectors, this number seems to be far from reality. In fact, over 70% of currently deployed IoT devices are in business, manufacturing, and healthcare sectors, and domestic and personal IoT devices seem to be concentrated only in the hands of enthusiastic early adopters, but not the general public. Usability is an important consideration for domestic and personal IoT systems that are deployed and maintained by non-professional users, as opposed to business, manufacturing, and healthcare sectors where teams of professional staff are hired to deploy and maintain large-scale IoT systems.

Unfortunately, building usable and integrated IoT ecosystems is particularly challenging for domestic and personal IoT systems because most low-cost IoT devices delegate the user interface to web- or mobile-based apps rather than using their own on-board interfaces, mainly due to form factor or cost constraints [54]. They usually require the use of a third device, most commonly the user's mobile device, to configure devices and establish a secure network. Ultimately, the user must be knowledgeable and determined to install, manage, and use a comprehensive set of IoT devices.

In the last two decades, as the IoT has inched closer to the goal of device and service integration, its widespread collection of data has come to be seen as a threat to individual privacy. It turns out that many people are uncomfortable with service providers collecting and archiving information about their daily routines and interactions, even if it is only for the purpose of improving service quality. Because interfaces are proprietary on most devices, users do not know what is going on behind the scenes, and they cannot be sure what service providers are doing with their data. Starting in 2017, a rash of revelations about misuse of data by some service providers has led many to assume the worst about the data handling practices of IoT service providers in general. The requirement to preserve privacy adds an additional dimension of complexity to the already difficult problem of IoT service integration. It sets up a tradeoff between privacy and service quality that increasingly feels like the uncertainty principle for IoT: an increase in service quality—which usually requires additional data collection—results in a proportional decrease in privacy. A long term goal for IoT researchers should be to demonstrate that service and privacy are not separate prizes in a zero sum game, but the aim of this thesis is more modest. The goal of this dissertation is to build technical solutions that can reduce the need for data siloing in IoT devices.

1.1 FOCUS OF THIS THESIS

The goal of this dissertation is to design and implement practical and scalable IoT building automation systems to meet the demands of the coming decades. To do so, it is important to first understand the landscape in which those systems

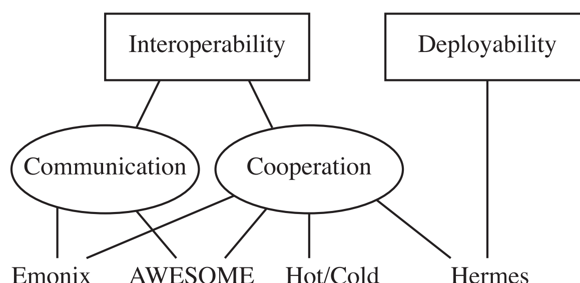

Figure 1.1: High-level goals of this thesis in relation to the individual projects that address them.

function. Since IoT systems are tightly coupled with their users, the approach in this thesis to deployment and coordination of devices necessarily conforms to user needs and expectations. One important challenge is that building occupants assume that building systems and resources are always available, so we must be careful not to inconvenience the users of these systems as we optimize them. Furthermore, seamless deployments of building-wide sensor systems often require nodes to operate on battery power for an extended period of time while having high enough performance to enable data processing and networking functionality.

Problem Statement In a crowded environment where IoT devices outnumber people, we seek to answer the following question:

> *How should we design IoT building management systems that minimize data and service siloing to optimize ease of deployment, quality of service, and interoperability of devices?*

In particular, this thesis explores techniques for horizontally integrating IoT and building management systems, even if they are built to be incompatible. Figure 1.1 depicts the goals of this thesis and how they are individually addressed by the systems that comprise it. Several important challenges to answering this thesis question are outlined below:

How to securely share data among multiple IoT systems? IoT systems generally use the client-server model to dispatch data management, user interfaces, and control to the cloud. In some cases, they use edge devices or cloudlets to store data locally. A few systems provide limited cloud-based APIs that give third-party services authenticated access to a subset of the data they collect. Each of these techniques puts a gatekeeper between potentially synergistic IoT devices, limiting their cooperation. The challenge is to increase the amount of data that can be shared among IoT services in a crowded local network that potentially has tens of devices coexisting.

This thesis presents a comprehensive suite of communication techniques that allows individual IoT devices to share data directly with one another on a local network. Emonix, an IoT electrical plug load monitoring system presented in Chapter 2, consists of many power monitors that can communicate directly on a local network to generate a comprehensive building-wide power consumption model in real time. Individual Emonix sensors can automatically discover one another and directly share data over a local network. AWESOME, an IoT water treatment monitor, communicates with Emonix devices in the same building to build an occupancy model based on water consumption and energy usage. Together, these systems can cooperate to generate a more complete view of activity within a building by sharing the information they collect. But building a model of activity within a building is not necessarily a straightforward task.

How to use shared data from multiple IoT systems to build a model of activity? Building automation systems typically monitor and control a single kind of building system: water treatment, HVAC, etc. The activity of these systems is typically linked through building occupancy, but building occupancy alone does not capture many important details about how these independent systems interact. The problem of inferring building occupancy from IoT sensor readings has been thoroughly explored, and its limitations are well understood [19, 21, 48, 49, 66].

Instead of employing a generic statistic like building occupancy, it would be preferable to use domain expertise to horizontally integrate IoT systems. This would allow them to share information about specific metrics that describe the

state of the building and the needs of its occupants. For example, a smart power strip may share electricity usage information with an IoT thermostat, allowing both systems to infer whether the building occupants are sleeping, awake, away from the house, etc. based on the historical temperature and energy data. The challenge is that it is not obvious, given a set of seemingly unrelated IoT systems, how to use data collected by one system to improve the performance of another.

This thesis demonstrates that Emonix and Hot/Cold can cooperate to control the energy used by a building's heating and cooling system. Sensors distributed throughout the building monitor the temperature in each room, and an IoT-controllable thermostat in combination with localized space heaters can be used to heat spaces within the home. Software running on a central controller optimizes an objective function that balances the costs of energy consumption and thermal comfort, sending commands to the distributed IoT devices as necessary. This building automation system uses the combined data from different IoT sensors (temperature, electricity, and air flow) to globally optimize an objective function that has competing terms proportional to each type of sensor data. It improves quality of service for the building systems in the sense that resource consumption is reduced while maintaining thermal comfort in the space.

How to implement our techniques on commercial IoT devices? With some notable exceptions, most commercial IoT devices are built with closed-source hardware and software without comprehensive third-party API support. Because they usually live behind firewalls and have limited compute resources, individual IoT devices are normally not called upon to support public APIs that would permit interactions with other unrelated services. A Nest thermostat, for example, cannot serve API requests directly to a third-party mobile app running on a user's smartphone. The app must send the request to Nest's cloud service which would in turn communicate through a proprietary channel to the Nest end device to request data or perform an action.

This architecture, depicted in Figure 1.3, follows naturally from the cloud services model that most IoT systems use, but its main disadvantage is that it is complicated and slow. A simple request—change the thermostat setting—must

be routed through the cloud, adding latency and divulging information about the user's behavior that the service provider does not need. Edge services may provide some relief by reducing the round-trip latency, but they do not offer a solution for horizontal integration among services; the siloing just happens in a different place. Introducing interoperability into existing closed-source systems is challenging because the data and commands are typically encrypted in transit to the cloud-based backend.

Hermes is a virtualization platform that allows IoT device software to run as a virtual machine on low-power end devices. This thesis demonstrates that it can be used to inspect I/O transactions as they transition from hardware to operating system. By intercepting I/O transactions, Hermes can control the information that is available to the stock software before that information can be encrypted for transmission on the network. The Hermes mobile device demo, presented in Chapter 5.3, demonstrates that Hermes can capture I/O events below the operating system's driver level and transmit them to a remote controller for processing. This technique would enable data marketplaces to generically capture data from IoT devices that do not support third-party APIs.

Our Solution We split the solution up into several components, depicted in Figure 1.2, comprising two broad categories of systems. The first, implemented by Emonix and Hot/Cold, monitors and controls the energy consumption of residential spaces. Our energy platform monitors and minimizes the energy used by plug load outlets and heating and cooling systems, which together account for between 60-70% of the energy consumed by an average residence. The second, implemented in AWESOME and SPOCK, monitors and controls water treatment systems. Because these two systems seem to be unrelated, they provide a good platform to help us understand how data is collected in silos. We use them to explore techniques for building horizontally integrated API services. Hermes, our hypervisor for microcontrollers, is a platform that allows us to unify data collection and services among multiple unrelated IoT systems.

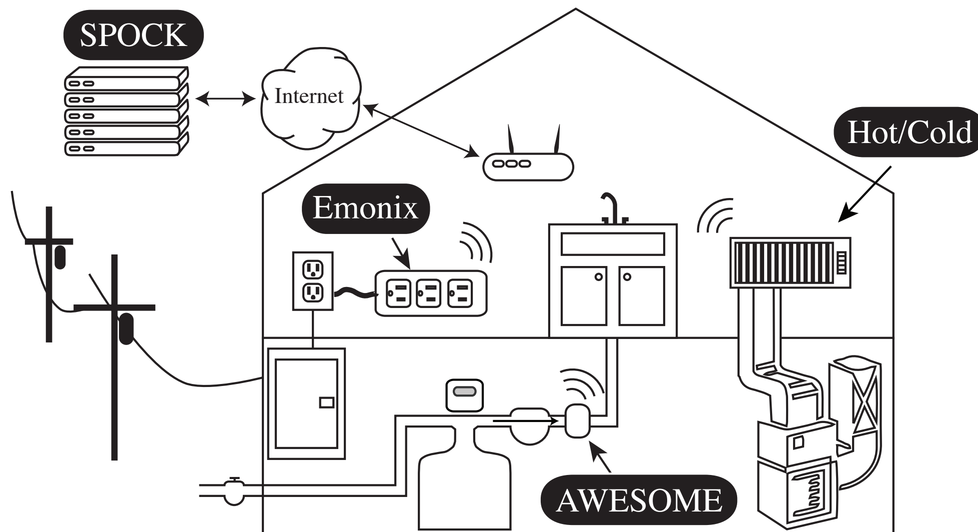

Figure 1.2: Overview of the building automation components presented in this thesis within a home.

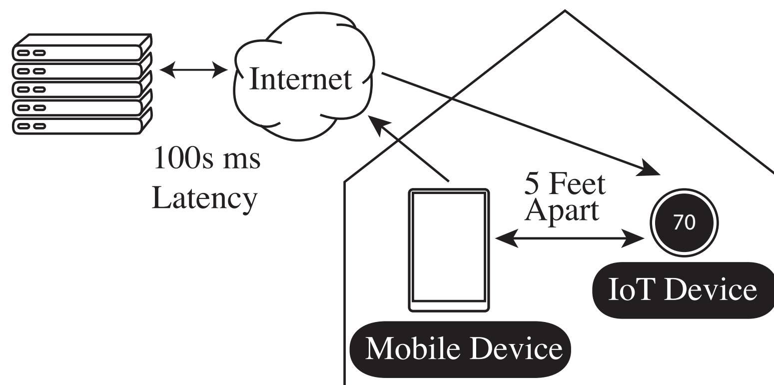

Figure 1.3: Cloud services model for implementing inter-device APIs.

Emonix

In the first part of this thesis, we present Emonix, a distributed, low-cost system for monitoring and analyzing energy consumption patterns in buildings. Emonix is designed with our custom energy sensing hardware and integrated communication units to be efficiently mounted in breaker panels of buildings.

In contrast to plug-based monitoring systems, this approach is less intrusive to users because it does not share their physical space, yet it still provides fine-grained real-time energy data in both space and time. The Emonix hardware platform is open, modular, and extensible. It provides an accessible data and configuration API for users, and we believe it is useful to the broad community. To demonstrate the usefulness of this platform, we deployed this infrastructure on two campus dormitories covering 60 rooms and 120 residents. In 2013-2014, we operated this infrastructure as an energy monitoring service for the residents for more than a year to help them understand their consumption patterns at different timescales. Our results indicate significant temporal variations in energy consumption patterns at different time scales, and that a small fraction of occupants can consume a disproportionately large amount of energy in such buildings.

We also developed a smart power strip version of Emonix that can monitor and disable connected loads. Together, these systems work well to reduce domestic electricity consumption, mostly by making users aware of their energy consumption patterns. Perhaps surprisingly, electric plug loads account for a relatively small fraction of building energy consumption (about 20%), so we looked for other large domestic energy consumers that could also be optimized.

Hot, Cold, and In Between

Forced-air heating and cooling systems frequently perform poorly in multi-story homes. Long, leaky ducts running from the heater or air conditioner to the rooms of the house can impede airflow to second story rooms, creating dramatic temperature differentials between different spaces in the house. Such temperature discrepancies can lead not only to occupant discomfort, but also to energy waste since the thermostat would have to be set higher to achieve the desired temperature in some rooms. In this work, we propose a decentralized platform for heating and cooling systems in medium-sized homes that aims to independently control the temperature of each room. The goal of this approach is to provide heating and cooling necessary for poorly conditioned individual rooms to reach the required comfort level. By heating and cooling

different rooms to different setpoints at different times, we can leverage our understanding about the way spaces in the home are actually used in order to fine-tune the environmental settings. We show that our techniques can both increase the residents' comfort and decrease the energy used by the house's heating and cooling system. In our primary test home, we estimate that our techniques could reduce natural gas consumption by roughly 18% during the coldest months of the heating season while increasing the temperature in the most-used living spaces by 2-5°F.

Awesome/SPOCK

In this piece of work, we introduce a control system we call AWESOME, which is a platform for improving efficiency in water treatment systems. We study water softeners, which are a primary source of Chloride ion pollution in effluent sewage water [15, 47]. When water softeners regenerate, they dump large volumes of salt and water down the drain. Existing water softeners use open-loop controllers that do not include water quality sensors to trigger regeneration. As a result, they generally regenerate too frequently, wasting salt and water and polluting the environment. AWESOME aims to reduce the frequency of water softener regenerations by sensing water quality and applying domain knowledge to trigger regenerations at optimal times. The sensor readings, which may be noisy or inaccurate, are processed by backend algorithms to accurately predict when regeneration is required. In a pilot deployment on the UW campus, AWESOME decreased salt consumption by an average of 27%, saving $5240 in the first year after installation.

We also develop a complementary cloud-based online control system called SPOCK, which forecasts future sensor readings based on historical trends. Using sensor value forecasting, we can plan control decisions for building automation systems hours in advance, with the goal of improving resource consumption efficiency. We test SPOCK on a dataset gathered by our water softener optimization engine [72] on the UW-Madison campus. We use sensor value forecasts generated by SPOCK to plan water softener regenerations up to 24 hours in advance. We compare the performance of SPOCK to an oracle

forecaster, which knows exactly what the sensor readings will be for all times in the future—this is possible because we are retroactively analyzing a dataset that has already been collected. For the dataset we analyze here, we demonstrate that SPOCK performs only about 2% worse than the oracle forecaster. Despite some errors in SPOCK's forecasts (compared to the oracle), it is still capable of making reasonable control decisions that are nearly optimal. SPOCK can reduce regeneration frequency—corresponding to reduction in wasted water—by 10% compared to existing algorithms for scheduling regenerations.

Hermes

Finally, we developed Hermes, a hypervisor for MMU-less microcontrollers. Hermes enables high-performance bare metal applications to coexist with real-time operating systems (RTOSes) and other less time-critical software on a single CPU. Hermes creates isolated virtual runtime environments for real-time tasks by adding a layer of abstraction between the hardware I/O devices and the software that services them. Virtualization on low-power mobile and embedded systems also enables some interesting software capabilities like secure execution of third-party apps, online privacy controls, and bare metal performance in a multitasking software environment. These features otherwise require additional hardware (i.e. multiple CPUs, hardware TPM, etc) or may not be available at all.

Reliability and programmability are frequently pain points of low-power IoT devices, which are often expected to live in inaccessible locations for many years without being serviced. These systems are difficult to debug because their *in situ* workloads are difficult to simulate—timing, in particular, tends to be a source of malfunction. RTOSes, which are normally used to manage multiprogramming on IoT devices, are not always able to respond quickly and deterministically enough to time-sensitive operations, particularly under high I/O load. We validate this observed timing problem by measuring interrupt latency in an RTOS environment and comparing to an experimental implementation of Hermes. In our evaluation we compare runtime performance of several realistic mobile apps on Hermes and FreeRTOS. We find that not only

is the interrupt latency lower in the virtualized environment, but it is also much more deterministic—a key figure of merit for real-time software systems.

1.2 CONTRIBUTIONS

The high-level contribution of this thesis is to design an extensible open-source IoT building automation platform that can interactively control multiple building systems with tightly integrated hardware and software. We demonstrate that it is possible to achieve horizontal integration among automation systems, reducing siloing and improving inter-system optimization. We also explore techniques to enable horizontal optimization on proprietary systems with software and hardware that we do not control.

Introduction of an Embedded Hypervisor for Improving Programmability of IoT Devices

The principal contribution of the Hermes hypervisor is to improve the programmability of microcontroller-based IoT systems. The main difficulty in IoT programmability is fault avoidance, which is still an open problem in the research community [28, 99, 101, 103] and in industry.

The CPUs in many IoT devices are microcontrollers, tiny highly-integrated single-chip computers that have the advantage of being low-power, low-cost, and easy to program. I/O processing in real-time systems based on these microcontrollers often suffers from unpredictable lag and jitter caused by other concurrent tasks that compete for CPU time on the device. For example, a single IoT device often balances processing loads from communication, data acquisition, user interface, and data processing tasks. Unpredictable inputs from one—for example, the communication interface—can create unpredictable behavior in others, which often causes malfunctions. Commercial mobile and IoT products are not immune to the problem. Instead of focusing on software reliability, which is often difficult to guarantee, commercial platforms dissect the software into trivially simplistic components that, in isolation, could not possibly fail. But in so doing, they make the hardware unnecessarily complex.

IoT hardware could be much simpler and smaller if all processing tasks could be performed by a single CPU.

Hermes, which we present in this dissertation, is the first hypervisor to run on a microcontroller with no memory management unit. By isolating different classes of software (communication, data acquisition, etc.) in different virtual execution environments, we demonstrate that Hermes can greatly reduce the propagation of timing uncertainty among tasks. We also demonstrate that Hermes can be used to intercept and filter I/O transactions as they traverse the path from hardware to driver to app. This feature allows us to implement custom APIs within Hermes that are not supported by the IoT device's stock software.

Integration of Building Automation Components

While some commercial building automation platforms provide multiple integrated IoT devices that can cooperate in a limited capacity, there is little work on broad integration of building automation systems that aim to optimize resource consumption across multiple building systems. For residential building automation systems in particular there is a lack of availability of coordinated building automation systems. Those that do exist on the commercial market tend to be one-off solutions that silo their data and do not integrate well with potentially complementary platforms. Until now, the opportunities that we can take advantage of with service coordination were not well understood. This thesis demonstrates that multiple IoT platforms can collaborate synergistically within a residential or commercial building to provide improved resource management. By sharing information at a device level with tightly integrated APIs, services can mine data collected by different IoT platforms to learn users' habits and optimize automation controls in a way that preserves privacy and improves comfort and efficiency.

Device-Level APIs

This thesis explores the use of device-level APIs, accessible within the local area network, that enable communication among nearby mobile and IoT nodes. Device-level APIs make communication faster and more private by eliminating the need to send data and commands through a remote cloud service. Emonix, Hot/Cold and Awesome all use device-to-device communication to enable low-latency controls. Hermes is a platform that enables device-level APIs to be implemented on third-party devices that do not normally support them. We demonstrate that by running the stock software as a guest within the Hermes hypervisor, we can filter and inject I/O transactions, enabling unsupported third-party devices to share siloed data and present a compatible device-level API to a larger system.

Although local communication that does not require cloud services had been implemented on some existing systems like Philips Hue and KeenHome, this thesis extends the idea by implementing it on multiple unrelated IoT devices. Our deployments demonstrate that local area device level APIs can improve latency and reliability in complex IoT systems.

Problem Validation

The problems we tackle in this dissertation have been carefully validated using passive measurements in real buildings, outlined below.

- **HVAC:** As part of our exploration of resource consumption of residential heating and cooling systems, we document significant discrepancies in temperature control of conventional forced-air HVAC systems that are common in US homes. We find that these systems are unable to control the temperature in all rooms of a home—some rooms are consistently warmer or colder than others. The problem is well-known among homeowners, HVAC technicians, and manufacturers. It is caused by friction between air and the ducts it flows through. Despite the widespread nature of the problem, there are few commercial solutions available.

- **Water Quality:** Most water treatment systems are operated by open-loop controls that do not sense water quality and respond in real time. Instead, they are configured once at installation and continue to operate with the same set of parameters. To account for unforeseen changes in water use patterns and incoming water quality, a maintenance technician is often required to verify that the system is working properly, usually a daily check that takes about 15 minutes. We deployed passive water quality monitors in several campus buildings for a period of nine months and found wide variations in the control settings for each. On average, we found that the systems were programmed to waste about 20% of their capacity, and some were programmed to waste as much as 50%.

  We found that installers routinely configure new water treatment systems under capacity because the lack of online feedback controls makes it impossible for the system to adapt to unforeseen circumstances in usage. Conservative configurations can be used to hedge against future deterioration in system capacity and other temporary fluctuations that would normally be handled transparently by online feedback controls. This practice incurs a burden of resource consumption and of facilities management time.

- **Electrical:** As part of our exploration in building resource management, we conducted a measurement study of the electric power usage by dormitory residents on the UW campus. We built and deployed custom power sensors that could independently measure the total electricity consumption of each room of a floor in residence halls. We found that in our dormitory deployment, electric power usage tends to be dominated by a few heavy users.

Evaluation in Residential and Commercial Testbeds

All four systems presented in this dissertation have been deployed and evaluated in commercial or residential buildings:

- We evaluate Emonix and Hot/Cold in a residential testbed consisting of two single-family homes with forced air HVAC systems that inconsistently condition the temperature within the spaces of the homes.
  - **Testbed 1** is a two story single-family residence with a forced air heating and cooling system with no programmable thermostat. We deployed five temperature sensors: two in bedrooms, one in the living room, one in a bathroom, and one outside. We used two Emonix smart power meters to control a space heater in one bedroom and a register booster fan in another.
  - **Testbed 2** is a two-store single-family residence with forced air heating and cooling system and a programmable thermostat. We deployed register booster fans and temperature sensors in the kitchen, living room, and three bedrooms. We did not use smart power strips to control fans or heaters in Testbed 2.
- We evaluate AWESOME and SPOCK in several mixed-use buildings, including three campus residence and dining hall, a hotel, and a midsize hotel. The campus buildings consist of approximately 100 dormitory rooms, shared laundry, and a large commercial kitchen. The hotel has approximately 30 rooms, laundry, and a small kitchen.
- We build an evaluation platform for Hermes that consists of prototype hardware and software environment that are typical of mobile systems. We write several custom apps for the platform, including GPS and camera apps. We also test third party software, including FreeRTOS and some of its demo applications. Our experiments with the Hermes mobile device demo indicate that Hermes is a practical runtime environment for mobile and IoT software that can improve app performance and data accessibility.

1.3 ORGANIZATION

The organization of this dissertation is outlined in Table 1.1. We first discuss the design of an IoT system for monitoring electricity consumption in buildings

Ch	Topic	Author's Related Work
2	IoT for Electrical Systems	BuildSys '13 [74]
3	IoT for HVAC	eEnergy '14 [70]
4	IoT for Water Treatment	BuildSys '15 [72], BuildSys '16 [73]
5	A Hypervisor for IoT Devices	HotMobile '18 [68], HotMobile '19 [71], IoTDI '19 [69]

Table 1.1: Organization of this dissertation and relation to the author's prior work.

(Chapter 2). We then discuss how our electricity monitoring platform can be integrated with a distributed network of temperature sensors and other HVAC components to independently control the temperatures in different spaces of a single-family residence (Chapter 3). Next we turn our attention to water treatment systems. Chapter 4 discusses the design and implementation of an IoT system for monitoring and controlling water quality in large multi-use buildings. Finally, in Chapter 5, we construct a software runtime environment that can be deployed on IoT devices to improve performance and interoperability.

2 EMONIX: AN IOT SYSTEM TO MONITOR ENERGY CONSUMPTION BY PLUG LOADS

Energy consumption of buildings has grown significantly and is a dominant contributor to global energy utilization. In the United States for example, energy consumption of buildings has risen by 48% since 1980, and it now represents 74% of the nation's electricity consumption [2].

The first step to control continued growth in energy consumption of buildings is to increase awareness among users and occupants. Understanding energy consumption patterns within buildings has, therefore, been a domain of continued research activity over the years [16, 31, 44]. In this paper, we report on the design and deployment of Emonix[1], a low-cost system for collecting and analyzing fine-grained electrical energy consumption patterns in buildings.

Why a new platform? When we started this project, we searched for a platform that would allow us to cheaply and easily deploy many energy sensors in a commercial building. We needed a platform that would easily permit software modifications to the metering devices. However, we were not able to find a commercial system with an open API that met our cost constraints. Table 2.1 lists estimated costs for deploying several commercial energy monitoring systems for a panel with 30 breakers. For each system, we included the cost of sensors, current transformers, power supplies, and other ancillary equipment.

Emonix design requirements With Emonix, we aimed to capture a holistic view of the energy consumption patterns of a large building. One challenge of managing a large building is that different areas are often occupied or controlled by small groups which may not cooperate or even communicate with one another. As such, it may be difficult for one individual or small group to understand the ways that resources such as energy, ventilation, and space are shared among building occupants.

[1] Emonix stands for Energy MONitoring and analytIX.

As a first step, we believe it is necessary to understand the electric power consumption of the building. However, we also acknowledge that there are other factors that contribute to a building's energy footprint. Our approach is therefore to design a system that has user-modifiable software which would allow us to easily extend the system's capabilities in the future. For this design, we identified the following requirements:

- **Open and accessible API:** Our energy sensor hardware should have an open and accessible API through which various monitoring tasks can be configured. For instance, it should be possible to instruct the sensors to collect energy samples at different granularities. It should also easily permit the addition of new features such as addition of network interface types or new sensor configurations.

- **Expandable sensor ports:** Each breaker panel often has a large number of branch circuits. Each sensor board should have a number of sensor ports to simultaneously collect energy measurements from multiple branch circuits. In addition, it should be possible to attach additional low-cost daughter boards to the main sensing board to flexibly increase the number of branch circuits being monitored. This approach will keep the costs of the overall system low.

- **Fine-grained reporting:** The energy sensors themselves should provide access to as much information as possible about the power distribution system. Ideally, this would include not just access to information about power consumption of a load, but also power quality metrics such as power factor and harmonics as well.

- **Flexible communication alternatives:** There are different alternatives to communication paths out of the sensing hardware. One could use some wireless technology, e.g., WiFi and ZigBee, or some wired counterparts, e.g., power line communication or even RS-232. We have found that none of these individual communication options works well in every scenario. Hence, a board which can be equipped with different communication alternatives is likely useful in diverse scenarios.

Product	Accessible API	Expandable Sensor Ports	Fine-Grained Reporting	Flexible Communication	Accessible Data
eMonitor		✓			✓
Modlet		✓	✓	✓	✓
TED		✓	✓		✓
Watts Up .Net			✓		
eGauge			✓		✓
Veris E30		✓	✓		✓
Emonix	✓	✓	✓	✓	✓

Table 2.2: Features of Commercial Power Meters. Emonix provides all features necessary in our deployment.

- **Accessible data:** The collection backend provides easy access to the measurements taken by the sensors. This may be as simple as exposing an API that allows us to query the database that stores data collected by the system.

There are many commercial products in the market that provide one or two of these features. Table 2.2 lists the features of a few of the meters we evaluated. Most of the energy monitoring systems we evaluated were targeted either toward plug-level monitoring (for single appliances) or utility entrance monitoring (for an entire building). In principle, it might have been possible for us to modify one of these off-the-shelf devices to monitor the power consumed by a single breaker instead. However, there were two hindrances to such an approach. First, none of the commercial meters provided adequate granular access to the energy data being collected. Second, each of these commercial systems would be very expensive to deploy at scale, primarily because most of the systems on the market are not designed to monitor many breaker panels or

individual circuit breakers. For these reasons a more cost-effective monitoring solution is possible.

Hence, we chose to architect a new system that is targeted specifically toward monitoring the power consumed by many branch circuits in a breaker panel. We were able to achieve dramatic cost reductions in the electric current sensing equipment because the overhead of the enclosure, microcontroller, power supply, etc. are amortized over many current sensors. Furthermore, we did not need to outfit every sensing device with an expensive wireless communication interface as is done by many outlet monitors because all of our sensors are located in the same room next to the breaker panel. As a result, an installation of Emonix could cost as little as $10 per breaker, while commercial meters such as Ted or Kill a Watt cost more than $50 per breaker.

Product	Cost
eMonitor [7]	$899
Modlet [9]	$1065
TED [8]	$3326
Watts Up .Net [12]	$7078
eGauge [3]	$1072
Veris E30 [11]	$6114
Emonix	**$330**

Table 2.1: Estimated cost of commercial energy sensors for a panel with 30 breakers. The cost for Emonix reflects only price and assembly. However, our devices are purchased at low volume, so our component cost are higher than they would be for large volumes. These figures include estimated costs for all components of the system, including sensors, current transformers, power supplies, etc. For Emonix, these figures only reflect hardware costs, while commercial products necessarily include costs of doing business such as technical support, engineering time, etc.

In addition, with Emonix, we aim to provide an *open* hardware and a flexible software platform[2] for energy research that is accessible to researchers, regardless of their hardware background. Most existing commercial platforms currently on the market do not expose any sort of API that would make it possible for experimenters to write and test custom software. By contrast, Emonix is fully implemented in the C programming language. Experimenters can modify any component of the

[2]Source code and design files available at
`http://research.cs.wisc.edu/wings/projects/emonix`

system, including the energy sensors themselves, through our open, clean, and well-documented interface. We believe this will allow others to build upon our initial successes.

Use cases through dormitory deployments: To demonstrate the usefulness of Emonix, we have conducted multiple energy monitoring pilots. For these pilots, we have worked with UW-Madison's Housing department. In particular, we have deployed Emonix in two different dormitories on our campus — Cole Hall and Chadbourne Hall. Among them, Cole Hall had a particular interest in our system because its residents are taking an active approach in reducing their energy footprint through sustainability clubs. In each dormitory, two students share a single room. Our deployments have covered 120 residents for a period of over four months, and provide some insights into energy consumption patterns in campus dormitories.

Key contributions: The following are the primary contributions of this work:

- We present our end-to-end electrical energy monitoring infrastructure, including our custom energy sensors for cost-efficient monitoring of branch circuits, with multi-modal communication capabilities, and various software components, e.g., databases and web-based dashboards.
- Our energy sensing hardware will be open, extensible, and flexible, and it implements an open API that allows users to add new modules to the system.
- To demonstrate usefulness of this platform, we have deployed them across multiple dormitories in our campus for a period of more than four months covering 60 rooms and 120 residents, and report on some interesting observations from the measurement study.

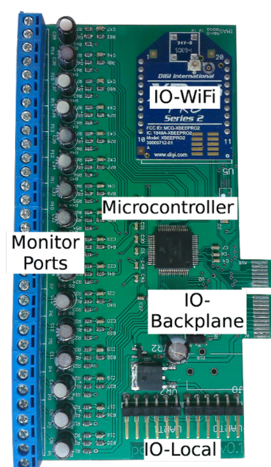

Figure 2.1: Custom energy sensor board. Monitors up to 16 current transformers and has three distinct forms of communication.

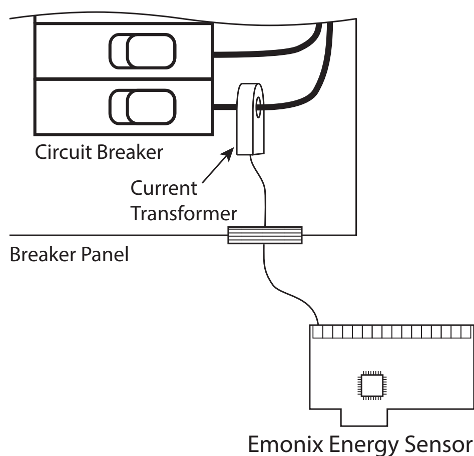

Figure 2.2: Connection of energy sensor board to monitoring device in breaker panel. Each branch circuit is monitored by a single current transformer, which is an analog device that can sense the current flowing through a conductor. The energy sensor board digitizes the analog signal generated by the current transformer and transmits the data to a gateway.

2.1 THE EMONIX ENERGY MEASUREMENT AND MONITORING INFRASTRUCTURE

Emonix is an embedded electric energy monitoring platform that can gather and analyze fine-grained energy consumption patterns at low cost and through minimal intrusion to the end user. Due to the unique constraints of our deployment environment, we chose to design custom energy sensing hardware and communication backend to deliver near real-time power consumption measurements to a database server. These measurements are made available to residents using digital signage in the building as well as a web-based monitoring dashboard with email updates[3].

Emonix Architecture

The monitoring system is a hybrid of multiple monitoring, communication, and storage technologies. The energy monitoring platform can be broken up into three primary functions:

1) Measure the power consumed by branch circuits,
2) Process the information as necessary for different applications
3) Relay the results to our database backend.

Custom energy measuring hardware:

The real-time sensing is performed by a Freescale ColdFire 51QE microcontroller. Our sensor boards use current transformers to sense the amount of electric current passing through each branch circuit in a breaker panel (see Figure 2.2). The microcontroller has more than twenty ADCs, allowing us to monitor many inputs on one board. Each analog input is pre-conditioned by a low-pass antialiasing filter that attenuates frequencies greater than 120 Hz. This ensures that high-frequency harmonics present on the voltage or current waveforms will not alias with lower frequencies. Since high frequencies typically carry a small fraction of the power consumed by an appliance, this does not reduce measurement accuracy by a large amount. In fact, calibration experiments done in our laboratory indicate that our sensors have less than 1% error in their power computations.

By default, each individual ADC is sampled at a frequency of 1.92 kHz (a harmonic of 60 Hz). The ADCs are sampled at high frequency to ensure that high-order harmonics that may be present in the current waveform do not alias with the 60 Hz fundamental. Harmonics above 1.92 kHz are attenuated by more than 80 dB by the analog antialias filter. After each 64 samples have been collected from each ADC, we apply a simple low-pass digital filter to the sampled waveform and downsample. We then compute the fast Fourier transform (FFT) of the downsampled waveform and transmit the magnitude

[3] Available to residents through a secure and private website.

and phase of the 60 Hz component. Using the API configuration interface, the above settings can be easily modified and even extended.

The communication interfaces on the energy sensor board were carefully designed to allow flexibility for various deployment scenarios. The sensor board is designed to provide an efficient monitoring solution for deployment environments with varying building wiring configurations, floorplans, and networking topologies. Figure 2.1 labels the three communication interfaces that the sensor board can use to relay the data it collects to a database server.

`IO-WiFi`: Communicate directly with an available WiFi Access Point to send the data to a server or gateway

`IO-Local`: Communicate directly with a gateway attached with an RS-232 serial cable

`IO-Backplane`: Attach to neighboring sensor boards on a *backplane* board to forward data to one gateway computer

In our residence hall deployments we take advantage of `IO-Backplane` in order to communicate with the gateway. It should be noted that while `IO-Backplane` looks very similar to a PCIe connection, it cannot be plugged into a computer's motherboard. Instead we have designed a "backplane" board, which presents PCIe slots (as seen in Figure 2.3). This allows up to five[4] sensor boards to share the same enclosure, power supply, and communication methods. These interfaces can also be used to load custom code onto the sensor board.

API Accessibility: Emonix also provides a complete open-source software set, including drivers for real-time data collection, and a simple RTOS that provides a POSIX-compliant interface to the programmer. Developers can compile custom software using `gcc` [67] and load their code onto a sensor board over the serial or WiFi interfaces. Emonix's API provides a subset of the POSIX calls that UNIX and Linux programmers are familiar with, including

[4]Five is our standard design, however a backplane board could be built to increase this size to practically any number of sensor boards

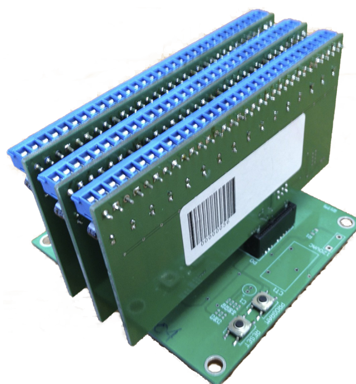

Figure 2.3: Multiple energy sensor boards connected to a backplane.

pthreads, memory allocation, and networking. Table 2.3 lists some of the features implemented by Emonix's API.

Using the Emonix API, we have written a few custom applications to run in place of the energy sensing software. In just an afternoon, we were able to completely replace the energy sensing software with an application that monitors and reports environmental data including temperature, humidity, and barometric pressure. The new application was multithreaded and took advantage of the Emonix network stack for exchanging messages with our database server. This exercise was a convincing demonstration of the power of our API.

Transport energy data: Once the data has been initially collected by the sensor board it must be packetized and transmitted to a gateway where it is aggregated, stored locally, and forwarded to a server for later use. In small buildings (single-family homes) with single breaker panels, a single gateway would be sufficient. In larger buildings such as the ones we currently have deployed in, a single gateway is unable to handle the entire distributed set of sensor boards and their continuous stream of energy measurements. Hence, we deploy multiple gateways which are responsible for collecting data from disparate sensor boards and forwarding them appropriately to the server backends. We use low-cost Single Board Computers (SBCs) that consume very little energy as gateways in

these deployments.

Process, store, display data: After the data has been aggregated by the gateway, it is transmitted to a database server. Once the data is on the database it can be stored for future use, or processed into a form more amenable for website viewing. The processing takes fine-grained data and computes commonly viewed data points (daily, weekly, and monthly energy consumption totals) for webserver access. The computation occurs once and is stored rather than each viewing needing to recompute the same data. This processing keeps database queries quick and minimizes latency for website viewing.

API Call	**Implementation**
`pthread` Library	Threads, Mutexes
`sleep`	Full
`time` / `stime`	Get/set UNIX epoch
`malloc` / `free`	Full
BSD Sockets	Partial

Table 2.3: Features supported by Emonix's API.

Flexibility and communication: Emonix is built to maintain flexibility in the face of unique electrical service layout, as is typical in commercial buildings of varying age. For instance, in one residence hall we are required to aggregate up to 90 breakers in a utility room. In another location, flush mounted breaker panels in common space would require our system to aggregate only 24 breakers and must reside above the ceiling tiles. Due to these varying constraints, we designed Emonix in a scalable way, which is useful to both an individual household as well as large dormitories.

The ability to provide as little as one and as many as 80[5] monitor ports allows Emonix to maintain a level of scalability unique among current energy monitoring systems. The cost-effective nature of our platform is due to several design iterations, which have allowed us to determine the most effective configurations.

[5] 16 monitor ports on each sensor board times five boards per backplane.

Generally we rely on a building's Internet connection to transmit data to the database server. However, many buildings do not always have reliable network connectivity in the utility rooms where breaker panels often reside (for both wired and wireless communication systems). In some cases, these electrical rooms have cinderblock construction that severely limits wireless communication in and out of these rooms. Due to these constraints we typically utilize a combination of communication techniques including: power-line communication (PLC), WiFi, Ethernet, and RS-232.

2.2 IMPLEMENTATION CHALLENGES

ADC Sampling

Real-time ADC sampling presented several unique challenges, most of which were timing-related. To get accurate results, the ADC sampling must be exactly right.

Phase Error. Since the ADCs that monitor the current and voltage must be sampled sequentially, there is a phase difference between the samples taken from each ADC channel. Phase error can create a problem when computing power, since the computation depends on the phase angle between current and voltage. We corrected for this problem by adding a different constant phase offset to the phase computed by the FFT before calculating the power. The delay between samples of our ADCs resulted in a phase offset of approximately 0.122π. To adjust for the phase error, we subtract $n \times 0.122\pi$ from the phase of each waveform, where n is the ADC index.

Real-Time Sampling. Since each of sixteen ADCs must be sampled at a frequency of 1.92 kHz, the processor must handle a total of 30,720 ADC conversions per second, each of which is processed by an interrupt service routine. Running at 60 MHz, this leaves only 1,953 CPU cycles per ADC interrupt for code execution, meaning that the CPU can execute roughly 500-1,000 instructions per ADC exception. To put this in perspective, a 64-point FFT

takes roughly 1 ms, or 60,000 CPU cycles on our processor.

Delay. In a related problem, we found that other interrupt service routines could delay the execution of the ADC sampling ISR, resulting in sample jitter. This would cause the energy in the 60 Hz component of the signal to spread to neighboring frequencies, resulting in inaccurate current and voltage measurements. To avoid this problem, we had to make the ADC sampling ISR the highest priority event in the system. Unfortunately, this requires us to take processing time away from communication and data processing tasks, which have their own real-time deadlines.

Integer Math

We chose to do all data processing with integer (fixed-point) math because floating-point support is typically expensive, power-hungry, and slow. All of our data is stored in 16-bit integers, meaning that we need just over 2kiB to store 64 samples from each channel simultaneously[6]. By contrast, IEEE floating point representation would require 4 kiB or 8 kiB depending on the representation standard.

However, integer math introduces more severe errors in the calculations than floating point math would. The errors are most pronounced in trigonometric functions such as sine and arctangent, which are used by the FFT. These errors are the primary contributors to the error in power consumption, and result in an overall error of less than 1%.

Safety

The national electric code in the United States requires power electronics to be certified by Underwriter Laboratories (UL) of another certified lab. UL standards specifically target equipment that comes in direct contact with conductors carrying line voltage (120V in the US), and are designed to minimize the risk of fire and electrocution. Most electricians will not install equipment

[6] $2\ bytes \times 64\ samples \times 17\ ADCs = 2176\ bytes$

that is not certified by UL for liability reasons. Our equipment is compliant with ANSI/UL 61010-1 and UL 796.

2.3 USE CASES IN RESIDENCE HALLS

We now discuss the data gathered from our deployment in the two dormitories over a period of several months (September to December 2012). There are various temporal trends, variations across residents, and numerous general energy consumption characteristics that we observed.

Thus far we have discussed the advantages of our platform monitoring at the branch circuit level, however there are a few caveats to note. First of all, while we are able to monitor each branch circuit individually, this does not guarantee that 100% of the energy consumed on the circuit belongs to a particular individual. For instance, if two residents in the same dorm room share a refrigerator, the current implementation of Emonix is unable to disaggregate and divide the energy consumed by the device between each roommate in a fair manner. This would also apply to roommates who may share computers, printers, etc.

Applications

The total data gathered is quite large and diverse, and it is not possible to report all aspects of it in entirety. Instead we will focus on some representative results. The dataset presented here corresponds to a set of 30 residents who have used our system the longest. We use randomly generated resident IDs to anonymize the different residents in this set.

Before we began the study, we took surveys of the building residents to try and understand the number and kind of appliances that were commonly used in a dorm room setting. The most common appliances we encountered were microwaves, refrigerators, laptops, and lamps. The housing department prohibits appliances with heating elements such as toasters and hot plates because they pose a fire hazard, so no residents reported these kinds of appliances. A lack of this class of power-hungry appliances would seem to

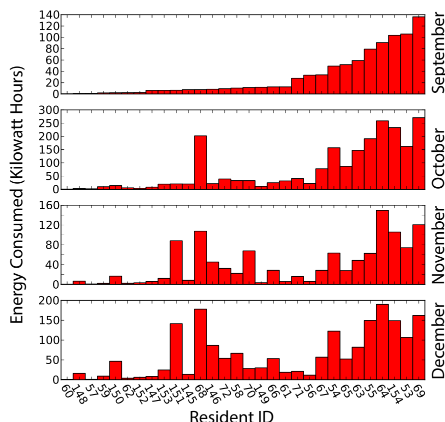

Figure 2.4: Total energy consumption per resident for the duration of our deployment for each month.

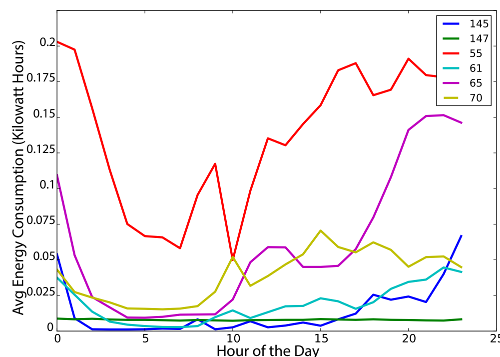

Figure 2.5: Average energy consumption as a function of hour of the day. Different lines correspond to different randomly chosen residents.

make university residence halls a fundamentally different type of deployment environment by comparison to other residential buildings.

Temporal behavior across months Figure 2.4 shows data we have collected across all four months during our study. The residents are sorted by their energy consumption amount in the month of September, and this ordering is maintained for the remaining months. As is generally true across typical user populations, there is a great diversity among residents in the amount of consumption. In particular, a disproportionately large amount of energy is consumed by a small percentage of residents. This is observed in each month of the study. This implies that energy conservation efforts may benefit most when focused on the small group of residents who manifest high usage patterns.

Previous studies of energy consumption patterns in large populations have found the distribution of energy consumption among users to be lognormal [75]. Our user population is not large enough to make that claim with any degree of statistical confidence, but based on the data we have gathered, it would not be surprising to find that the data were in fact lognormally distributed.

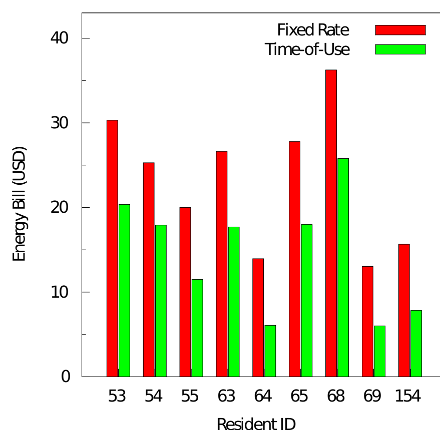

Figure 2.6: Hypothetical energy bill for several residents.

We can also see that the energy usage behavior of many users stay consistent from month to month. However, this trend is not universal, e.g., residents 68 and 151. Finally, as expected, total energy consumption grows from September to December as the seasons change.

Behavior across a day Most residents follow very consistent daily patterns as is evident in their average energy consumption for each hour of the day in Figure 2.5. There is a consistent trend which indicates that most users do not maintain the same level of energy consumption for all hours of the day.

Time of Use vs. Flat-Rate Billing The data we collected from dormitories using Emonix was used by another group on our campus to evaluate electric utility billing policies. Our local electric utility offers its customers two billing rate options: fixed rate and time of use. Under fixed-rate billing, utility customers pay the same rate for the energy they consume at all times of the day, plus a daily customer charge. Under time-of-use billing, customers pay a reduced rate (about 50%) for electricity consumed during off-peak hours (9PM – 10 AM daily) and an increased rate (about 160%) for electricity consumed during peak hours.

Figure 2.6 shows the hypothetical electricity charges for one week of energy consumption for several of the biggest energy users in this study. These figures are based on our local electric utility's billing rates [6]. The red bar shows each resident's hypothetical bill under a fixed energy pricing scheme, which is the scheme used for most residential utility customers. The green bar shows the energy bill for time-of-use billing, in which the utility charges higher rates during peak hours and lower rates during off-peak hours. We found that everyone we studied would benefit in some degree from using the time-of-use billing scheme. We attribute this to the propensity of college students to stay up late at night, thereby shifting their peak usage to later hours. In fact, many of the students who participated in this study are likely to be absent from their dorm rooms until late at night. Figure 2.5 clearly indicates that resident 65 routinely uses most of their energy between 8PM and midnight. This type of behavior would result in considerable savings under the time-of-use billing scheme.

2.4 SUMMARY OF EMONIX

Emonix is an energy monitoring and analytics platform that enables us to track electric plug load energy usage in a building at the level of individual rooms. We deployed Emonix in several residence halls on the University of Wisconsin campus and found that informing the building occupants about their energy consumption helped them to reduce their overall usage. But the reduction in energy consumption we were able to achieve was only a small part of the overall building consumption. Other building systems like hot water and heating and air conditioning account for a comparatively large portion of the building's overall energy footprint. We now turn our attention to energy used by heating and air conditioning, which accounts for roughly half of overall building energy usage nationwide.

3 HOT, COLD, AND IN BETWEEN: AN IOT PLATFORM TO CONTROL HVAC SYSTEMS FOR COMFORT AND EFFICIENCY

Residential heating and cooling systems frequently have trouble maintaining a constant temperature distribution throughout the building, particularly in multi-story homes. Homes heated by forced air systems are often the most difficult to control because conditioned air is not always distributed equally among all rooms in the house. In general, the farther away a room is from the furnace, the less conditioned air it will get. However, most residential furnaces have a single, centralized point of control – the thermostat – which controls delivery of conditioned air to the entire house. Thermostats are often located close to the furnace – generally on the first floor of a two story house – in rooms that receive a relatively high flow of conditioned air from the furnace or air conditioner.

Since there is generally only one point of sensing and control in most residential heating and cooling systems, the heating and cooling system can have an incomplete view of the temperature distribution in the house. Furthermore, residential HVAC[1] systems do not have the ability to fine-tune the temperatures on a room-by-room basis because there is no way to selectively heat or cool a subset of the rooms even if the thermostat knew that there were temperature disparities in some rooms. Because they do not have fine-grained control for multiple rooms, residential HVAC systems often spend a lot of energy to condition unoccupied areas of the home. For example, the bedrooms are frequently located on the second floor of a two-story house. These spaces are often too hot during the summer months and too cold during the winter months because they are not receiving enough conditioned air from the HVAC system. At night, the temperature in the bedrooms can be high or low by as much as ten degrees Fahrenheit, making it difficult to sleep, while the unoccupied areas such as the kitchen and living room are relatively comfortable.

To address this problem, some larger houses can be outfitted with multi-zone HVAC systems. In these systems, the home is divided into regions or

[1]Heating, ventilation, and air conditioning

zones that each have independent sensing and control. This is often done by outfitting each zone with an independent furnace, air conditioner, and duct system. For new construction, multi-zone HVAC systems are much more expensive to operate than single-zone systems because multiple furnaces and air conditioners may need to be installed and maintained. It is also extremely expensive and intrusive to retrofit an existing home with a multi-zone HVAC system because in addition to equipment costs, ductwork needs to be redone, requiring drywall to be torn out. The additional installation and maintenance costs put multi-zone systems out of reach for most homeowners.

Inefficiencies in residential heating and air conditioning systems can lead to discomfort in the home and excessive energy consumption. Since approximately 70% of US homes are conditioned by forced-air AC systems, and approximately 62 % of US homes feature some form of warm air furnace, these problems have broad reach across many people [41]. Furthermore, the US Department of Energy estimates that more than half of the energy consumed by buildings in the United States is used for space heating and cooling [2]. For this reason, small inefficiencies in a single type of system — the furnace or air conditioner — can have a massive effect on the country's energy consumption, while low energy prices insulate individual consumers.

To address these problems, we augment existing forced-air heating and cooling systems with a few additional components to provide localized control over the temperature of individual rooms. The relevant system components are depicted in Figure 3.1.

1. **Network-connected temperature sensors** located in every room of the home that measure environmental conditions and report to an off-site database server.

2. **Register booster fans** increase airflow from the heating and cooling systems to rooms that are too cold in the winter or too hot in the summer.

3. **Space heaters** locally augment the central heating system in rooms that are particularly cold.

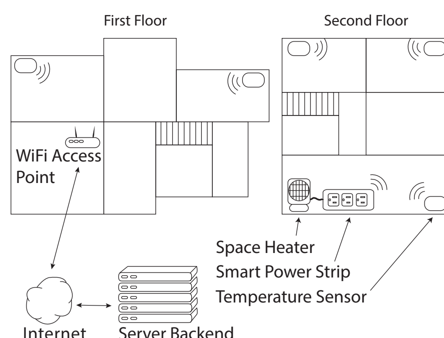

Figure 3.1: Floorplan of Home A in our study, showing the placement of temperature sensors throughout the building. Each sensor connects to the Internet through an existing WiFi access point. A remote server collects the data and controls local heating and cooling appliances through network-connected smart power strips.

4. **Network-connected smart power strips** with embedded relays that can control each register booster or space heater by switching its power on or off.

5. **Intelligent server-side control software** that analyses data reported by the environmental sensors and sends control messages to the smart power strips to regulate the temperatures of each room in the home.

Using these components, we can make residents more comfortable by individually controlling the temperature of small areas of the home. This is beneficial to homeowners and occupants in two ways. First, localized environmental control allows us to eliminate hot or cold regions of the home by selectively conditioning individual rooms. Second, we can avoid over-conditioning regions of the home that are not actively used.

Fortunately, heating and cooling systems lend themselves well to technological solutions for controlling energy consumption because users are accustomed to these systems being under automatic control. A reasonable approach to this problem would therefore be to develop control techniques that take energy consumption into consideration. By taking a global view of the temperature distribution in the home, we seek to control the temperature in a building on

a per-room basis. This will allow building occupants to condition spaces in a way that is more closely aligned with the way the space is actually used.

The approach proposed in this paper allows homeowners to cheaply and easily retrofit an existing house that has single zone HVAC with fine-grained multi-zoned control. Our system is much easier to deploy than multi-zone HVAC because it can be installed by homeowners without the burden and expense of hiring a contractor. Furthermore, it can effect control of the HVAC system on a much finer granularity than multi-zone HVAC.

We demonstrate that local temperature control is practical for average-sized two-story single family houses. To our knowledge, previous work in this area has focused primarily on small single-story houses, which generally do not have large temperature disparities between different areas of the building.

The goals of this work are to identify the sources of inefficiency in existing forced air heating and air conditioning systems and develop technical solutions to mitigate these problems. To this end, we have identified two related objectives. First, we need to make sure that the heating and air conditioning systems are correctly performing the tasks they set out to accomplish. If a user sets the thermostat to 70 degrees, we need to make sure that the temperature is actually 70 degrees. Using a decentralized sensing and control system, we are able to individually adjust the temperatures of the rooms in a building to ensure occupant comfort. This is important because we want to discourage residents from overcompensating for local temperature discrepancies (in a single room for example) with global corrections like turning the temperature setpoint up several degrees in the entire house.

Second, we wish to reduce the energy consumed by HVAC systems by intelligently conditioning spaces in the home based on the occupants' needs. For example, rooms that are heavily used should be tightly controlled at a comfortable temperature, while rooms that are not as frequently used may be allowed to drift. By combining these two techniques, we expect that it would be possible to set back the global thermostat settings while selectively conditioning rooms that are heavily used. In doing so, we can save energy and make the residents more comfortable.

The primary contributions of this work are as follows:

- We develop and deploy a distributed environmental sensing platform that can monitor the temperatures of each room in a home and report those measurements to a central controller.
- We use measurements taken by this system in *several homes over the course of six months* to identify sources of inefficiency in the HVAC systems. We observe consistent variations on the order of 5-10°F in second-story rooms among the houses we studied.
- Using feedback control on a room-by-room basis, we show that it is possible to simultaneously optimize the temperature distribution in a home while reducing the energy consumed by the HVAC system. In our testbed, we estimate that homeowners can reduce their natural gas consumption by 18% while reducing temperature variations in the rooms they use most.

3.1 UNDERSTANDING EXISTING HEATING AND COOLING SYSTEMS

In this section, we discuss the way existing heating and cooling systems operate in the homes we studied. For this work, we deployed temperature monitoring equipment in each room of three homes, which we call Homes A, B, and C. The homes we studied for this work were all two story single-family houses with single-zone forced air HVAC systems whose owners complained of temperature discrepancies between the first and second floors.

Shortcomings of Centrally-Controlled Heating And Cooling Systems

Each home we studied exhibited roughly the same types of behavior with respect to its heating and air conditioning system. Each had a 6-10°F temperature difference between the first and second floors during the warmest and coolest months of the year. We found that the airflow from the HVAC system was dramatically different on the first and second floor. Table 3.1 shows our measurements of airflow coming from the vents on the first and second

floors of each home. Each had about a 50% reduction in airflow coming from the vents on the second floor, resulting in a commensurate reduction in conditioned air mass available to heat or cool the rooms. The difference in airflow between the two floors is a plausible explanation for the temperature differences.

Every house we studied had its thermostat in the dining room on the first floor. In two of the three houses, the residents did not even use the dining room on a regular basis, so it made little sense for the HVAC control system to be taking its only temperature measurements in that location. Since the point of measurement was on the first floor close to the central HVAC system, the whole first floor in all three houses roughly followed the schedule that the residents programmed into their thermostat. However, the second floor consistently had 5-10°F temperature variations. In order to keep the second story comfortable, the thermostat would have to be programmed to over-heat or over-cool the first floor. To compensate, the residents generally chose to program the thermostat with some middle ground settings that kept the first floor slightly over-conditioned while the second floor was under conditioned, but liveable. For example, the programmable thermostat in Home C was set to 68°F during the evening hours while the residents were working in the second story office. The residents reported that they were comfortable between 60-65°F, but they increased the setpoint throughout the home in order to be comfortable on the second floor.

Figure 3.8 shows temperature measurements taken over the course of an average day in December for Home C. We refer to the difference between the setpoint of the thermostat and the actual temperature in a room as the *temperature error*. On the second floor, the master bedroom is consistently 6-8°F cooler than the dining room on the first floor during the time period shown, making that space uncomfortable for the residents.

Home	1st Floor	2nd Floor
A	2.9 m/s	1.4 m/s
B	3.1 m/s	1.6 m/s
C	2.8 m/s	1.4 m/s

Table 3.1: Airflow measurements taken at each home in our study. There was roughly a 50% decrease in conditioned airflow to the second floor.

Rooms on the first floor can also suffer the effects of unequal airflow. Spaces located near the furnaces tend to receive disproportionately high air flow, resulting in highly variable temperatures. This is because conditioned air from the furnace or air conditioner has to travel through a shorter length of duct which puts less back pressure on the air flowing to these spaces.

Since gas fired furnaces produce conditioned air at temperatures of 130-160°F, rooms close to the furnace can receive high volumes of extremely warm air that has not lost heat while traveling through ducts. The result can be uncomfortably warm and highly variable conditions in these rooms. Figure 3.2 shows a plot of the air temperature in a room that is close to the furnace and receives a lot of over-conditioned air. The measurements shown in the figure were taken on the opposite side of the room from the HVAC register, so the sensor experiences the same hot and cold fluctuations as the rest of the room.

In this work, we seek to identify the reasons for large temperature errors and to take steps to correct those errors in places where they cause the residents discomfort. More specifically, our goals are to

- Provide additional heating and cooling to rooms with large temperature errors in order to make those spaces more comfortable.

- Allow the user to set different setpoints for different rooms at different times of the day so the living space is conditioned to meet the user's needs.

For example, we found that residents will often heat or cool their entire house so they can be comfortable in just one or two rooms. This represents an enormous waste of energy that can be avoided by ensuring that the actual temperature in a room adheres to the desired setpoint schedule.

A Distributed Environmental Sensing System

In order to understand the temperature discrepancies between rooms' setpoints and their actual temperatures, we developed and deployed embedded temperature and humidity sensors in each room of several single-family houses during

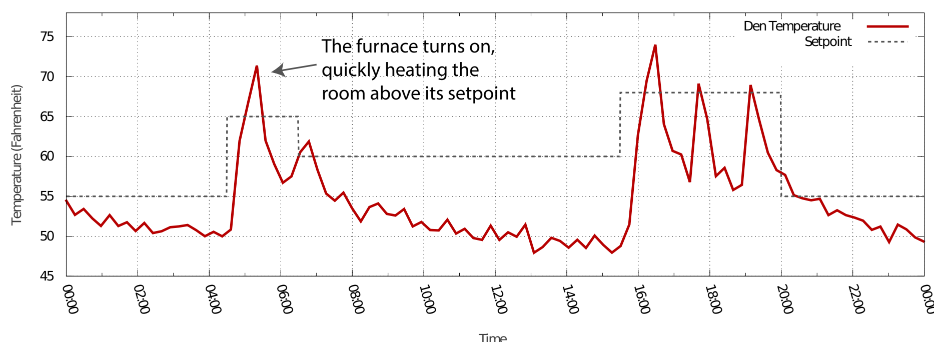

Figure 3.2: Without correction, even rooms close to the furnace or air conditioner can experience wildly fluctuating temperatures. In the den of Home C, high-temperature air from the furnace quickly heats the room above its setpoint. When the furnace turns off, heat leaks away through the three exterior walls.

the course of a six month period. We chose to develop our own hardware platform because it is cheaper and much more configurable than commercial off the shelf components. Figure 3.3 shows a photo of the sensors as they were deployed. The hardware and software are variants of the Emonix platform [74], which is a flexible network-connected sensor platform that can be easily reconfigured for various distributed sensing applications.

Each sensor connects to the Internet through the home WiFi network and reports its readings to a central controller at one-minute intervals. We chose to connect our sensors to the Internet using WiFi because it is widely available in residential environments. By contrast, many embedded sensor networks use an intelligent bridge to facilitate communication between a low-end sensor network (eg. Zigbee) and an off-site controller. While this approach can slightly simplify the embedded nodes, it has the disadvantage of increasing the complexity of the overall deployment by adding a new point of failure (the intelligent bridge).

Our nodes collect temperature readings once per minute and transmit them to a centralized off-site controller. The controller logs each temperature reading in a database and returns an application-level acknowledgment to the sensor. Sensor readings are stored locally on the sensor nodes, and messages are retransmitted in case of packet loss.

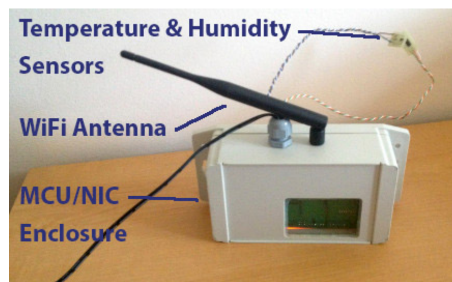

Figure 3.3: Photograph of temperature sensors used to monitor the environmental conditions in each room.

Figure 3.4: Photographs of our control equipment. (a) A register booster fan controlled by Emonix smart power strip. (b) An Emonix power strip controls a space heater that provides additional heat in rooms that are too cold.

3.2 AUGMENTING THE CENTRAL HVAC SYSTEM WITH LOCALIZED CONTROL

In order to reduce temperature error in second-story rooms, we added local temperature controls to a subset of the rooms on the second floor of Home C that were most heavily used by the residents. In addition to setting a global (house-wide) temperature schedule on their programmable thermostat, we were able to provide per-room temperature schedules in these rooms. We assume that occupancy data and analysis techniques are available to evaluate the residents' habits and construct a thermostat schedule that meets their needs. Several existing pieces of work — including commercial products — address this problem [14, 82, 100].

For this work, we allow the thermostat to maintain control over the global temperature settings for the home, and we install additional devices to augment the central HVAC system on a per-room basis. Each room in the home that has localized control installed can have an independent thermostat schedule

separate from the main HVAC system's schedule. This requires minimal installation on the part of the homeowner, in keeping with our goal of making it possible for homeowners to do the installation themselves.

We used several techniques to locally adjust the temperatures in each room: register damper control, register booster fans, space heaters, and window air conditioners. Register dampers can be closed to shut off air to areas of the home that are properly conditioned. Likewise, register booster fans can help push air through the ducts into rooms that are not receiving enough air flow to help pull air from the duct system. Space heaters can be used in rooms where the central HVAC system can not sufficiently heat or cool the space, even with the addition of localized register controls.

Except for the register dampers, each of these techniques uses a device that can be controlled by an Emonix smart power strip. The smart power strip has two key components used by our controller. First, the controller can use an internal relay in the smart power strip to actuate the devices that draw power through it. This allows us to easily turn electrical equipment on or off by simply sending a command over the Internet to the power strip. Second, the controller can log the power consumed by the devices connected to the power strip. This allows us to monitor the energy used by our system for the purposes of assessing energy efficiency. Each technique is discussed in detail below.

Control Algorithm

The goal of our control algorithm is to minimize the temperature error in rooms that have local control equipment. Our controller takes as input a temperature schedule for each room in a single family house as well as a set of temperature measurements taken in the room. Two example temperature schedules are shown in Table 3.2. Based on the room's average temperature measured over the last five minutes, the controller can either turn on or turn off a power strip which controls the registers, space heater, etc. in the room. The controller has 2°C of hysteresis around the setpoint.

If the measured temperature in a room under local control is more than 2°C below its local setpoint[2], the central server turns on a space heater or register booster in that room. If the temperature is more than 2°C above the room's local setpoint, the controller turns off the local control devices. The localized heating in the room will not turn back on until the room stabilizes at 2°C below the setpoint.

The temperature in each room under local control is re-evaluated once every minute during the early morning and evening hours. These are the times when the residents in the home are most likely to be using the second story space. During the middle of the day and at night, the local control mechanisms are turned off because the residents are either asleep or away from the house. While the local control mechanisms are turned off, the central furnace system is responsible for maintaining all spaces in the home at a global setpoint. Control actions such as turning space heaters on or off can also be taken once per minute in response to each new temperature reading, but in practice the switching is much less frequent.

```
Data: Temperature Readings
Result: Device state for local heating
foreach room under local control do
    if during household's normal occupancy hours then
        read temperature;
        if temperature below setpoint then
            turn heater on;
        else
            turn heater off;
        end
    else
        relinquish control to thermostat;
    end
end
```

Algorithm 1: Localized temperature control, run once per minute on the controller.

[2] The local setpoint in a room may be different from the main thermostat's global setpoint.

Register Control

Our first approach to provide localized control for the HVAC system in a home was to selectively close registers that feed parts of the house that are already well-conditioned. A similar approach is commonly used by multi-zone HVAC systems, in which a damper[3] is used to shut off air flow to entire lengths of ductwork that feed sections of the home. In this work, we chose not to use dampers because they are expensive and intrusive to install.

We believed that since all rooms on the first floor of the test houses in our deployment were receiving adequate air supply from the furnace and air conditioner, we could redirect air to the under-served second floor. We tried closing the registers on the first floor manually using the levers on the vents, and in some cases, we used cardboard and duct tape to seal off the duct. Similar techniques have been shown to work in smaller single story homes by other studies [102]. However, this did not have a detectable effect on second story air flow or temperature in any of the three homes we studied. Since our primary goal was to improve airflow and temperature variations in second story living spaces, we could not rely on register control alone.

In some cases, register control was useful to temper the wide variations we observed in first story rooms. As depicted in Figure 3.2, first-story rooms can often experience dramatic variability in temperature, particularly in rooms located near the furnace. By closing some vents in strategic locations on the first floor, we were able to reduce many of these variations.

We believe that the ducting system in these houses had leaks that allowed air to escape from the system before it reached the second story rooms. The US Department of Energy estimates that between 25-40% of conditioned air in average forced air heating systems escapes through leaks in the ducts [1]. In fact, they estimate that even well-sealed ducts will leak up to 20% of the conditioned air that flows through them. Since the ducts feeding rooms on the second floor are physically longer than those feeding the first floor, there is more opportunity for conditioned air to escape on its way to the register.

[3] A damper is a valve that controls airflow through a duct.

Figure 3.5: The internals of an Emonix smart power strip include a relay to switch power to the plugs it serves and an analog adapter board to sense the power consumption of the connected appliances. The main CPU sits beneath the analog adapter board along with its WiFi interface that directly connects to the home's network. This allows energy measurements to be relayed to an off-site controller.

Furthermore, friction between air and the wall of the pipe causes the air to slow down as it passes through a long pipe.

The perils of closing off the registers that feed large portions of a home are well-known to HVAC contractors and academics alike [104]. Increased back pressure caused by closing off registers can damage some components of the system. The resulting decreased airflow can also reduce the HVAC system's ability to heat or cool the house — decreasing airflow to one portion of the house does not necessarily increase flow to other areas. For this reason, systems that rely exclusively on dampers to direct conditioned air to specific areas in the building may have to allow some air to flow to spaces that do not need it in order to avoid excessive back pressure in the ducts.

Because we were not able to satisfactorily improve temperature discrepancies using register control alone, we tried using other measures.

Register Booster Fans

In a second attempt to increase air flow to poorly-conditioned rooms, we installed booster fans in the registers of some rooms. Figure 3.4 (A) shows a photo of a booster fan installed in Home C. These fans pull air out of the registers, selectively creating lower pressure at the output of the duct system

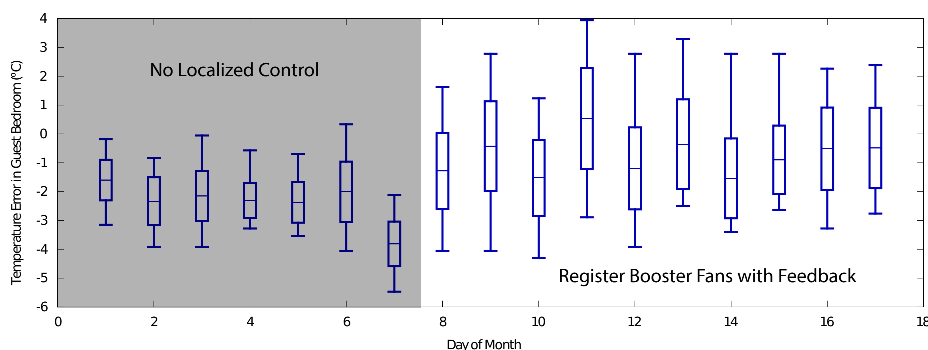

Figure 3.6: Deviations from the setpoint temperature in the guest bedroom of Home C. Before installing register booster fans, the room was on average 2.5°C below the thermostat's setpoint. Booster fans decreased the average deviation to 1.1°C below setpoint.

in places where conditioned air was needed. The booster fans we used take only about five minutes to install, and the only tool required to do the job is a screwdriver.

The booster fans we used increased the air flow to second story rooms to roughly 2.5 m/s while drawing only about 5 Watts of electric power. As a result, we observed a corresponding improvement in temperature in rooms that had register boosters.

One drawback of using register booster fans is that they can only work while the central furnace or air conditioner is on. This is also true of damper actuators as employed by multi-zone HVAC systems. Decentralized HVAC control systems that rely only on damper or register controllers would have a difficult time heating or cooling a small portion of a home.

We tested booster fans in two rooms of Home C. The guest bedroom, which has only one small section of exterior wall and its own return, responded well to the boosters. The master bedroom, with three large outside walls and no cold air return, did not benefit from boosters. Figure 3.6 shows a box plot of the temperatures errors in the guest bedroom during the first half of the month of December, before and after register boosters were installed.

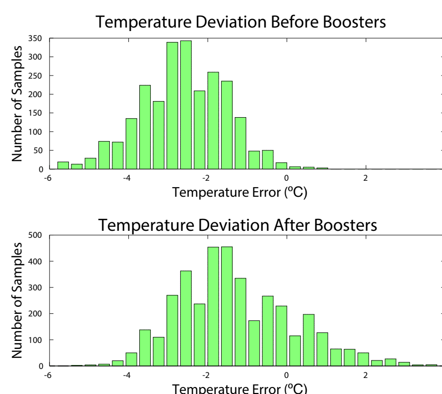

Figure 3.7: Histogram of temperature deviations from the setpoint in the guest bedroom of Home C before and after the installation of register booster fans. The use of register booster fans decreases the magnitude of the average temperature error by 1.4°C.

Figure 3.7 shows a histogram of the temperature error in the guest bedroom before and after booster fans were installed. Before installation of register booster fans on the 8th day of the month, the temperature was on average 2.5°C below the setpoint in the guest bedroom. Ideally, our goal was to increase the temperature in the guest bedroom so the mean deviation from the setpoint is zero. This would mean the histogram in Figure 3.7 would center around 0. After installation, the average temperature deviation was about 1.1°C below setpoint. This improvement corresponds to about 50% of the achievable goal. Intuitively, an increase of 1.4°C as we saw in the guest bedroom has a noticeable effect on comfort.

The standard deviation of temperature errors also increased after installing register boosters. This is likely because the air coming out of the registers is much warmer than the air in the room. Before installing register boosters, there was less airflow from the vents, and that air was not mixing well with the air in the room because it was moving slowly as it exited the vent. After adding register boosters, the warm air would mix more with the room's air, and the temperature in the room would fluctuate up and down as the furnace cycles on and off through the course of the day.

Unfortunately, register booster fans do not work well for all rooms. If a room is poorly insulated or particularly far away from the furnace or air conditioner, the additional airflow they provide may not be enough to correct temperature disparities. The master bedroom in Home C is one such case. Not only is it poorly insulated and physically far away from the furnace, it also lacks a cold air return. Cold air returns are used to evacuate stale air from the space so that excessive pressure does not build up, allowing air to circulate more freely throughout the house. Thus, the lack of a cold air return can lead to inefficiencies in the HVAC system.

Space Heater

For additional heating capacity in the winter, we added a small space heater in the master bedroom that could be controlled by a smart power strip. If the controller detects that the the temperature in the master bedroom drifts below its setpoint, it can turn the space heater on by sending an asynchronous control message to the power strip. This system could be logically extended to also control actuation of an air conditioning (cooling) unit in the hot summer months.

Figure 3.4 shows a prototype of this configuration as deployed in Home C. The final system would be more compact, with the control and communication for the space heater embedded in the heater itself. In-wall space heaters could also be used to save space and reduce cord clutter, though this would require more complex installation.

Figures 3.8 and 3.9 illustrate the improvements in temperature errors in the master bedroom from using localized control. Figure 3.8 shows the temperature profile of the master bedroom over the course of a typical day before installing local control in the master bedroom. The temperature in the master bedroom is roughly 5°F below the dining room (where the thermostat is located) and it is 10°F below the setpoint for most of the evening hours. Figure 3.9 depicts the situation after adding localized control. During two crucial periods — early morning and mid evening — the master bedroom temperature is still slightly below setpoint, but it is actually warmer than the dining room.

	Heating °C Days	**Natural Gas Usage**	**Electric Heating**
Furnace Only	804	107 Therms	0 kWh
Local Control	659	87 Therms	54 kWh
Improvement	18%	20 Therms = 586 kWh (**18.7%**)	(54 kWh)

Table 3.3: Energy savings that resulted from the use of localized control in Home C.

3.3 ANALYSIS

Time Period	Before	After
5:00 AM - 8:00 AM	65 °F	60 °F
8:00 AM - 4:30 PM	60 °F	55 °F
4:30 PM - 10:00 PM	68 °F	60 °F
10:00 PM - 5:00 AM	55 °F	55 °F

Table 3.2: Set points of the programmable thermostat in Home C before and after adding localized control.

In Home C, we used localized techniques to make the second story rooms more comfortable. We chose Home C as the primary subject for active control, because out of all homes we studied, Home C best exemplified the common problems associated with current HVAC systems. Our main goal was to regulate the temperature in the master bedroom between 60-65°F in the evening while the rest of the house cooled down. This is the room where the occupants spend most of their time in the evening, and it was also the room that had the biggest temperature errors when controlled only by the central heating system. In this work, we assume that a suitable thermostat setpoint schedule can be established either manually or using automated techniques that take account of occupancy data as well as user input. Several pieces of work have demonstrated that this is feasible [51, 82, 100].

Prior to adding localized control, the master bedroom had been so cold that the residents only used it for sleeping. This was inconvenient because it also served as an office, so the residents had to move their computers, desk, and files to other areas of the house that were more comfortable.

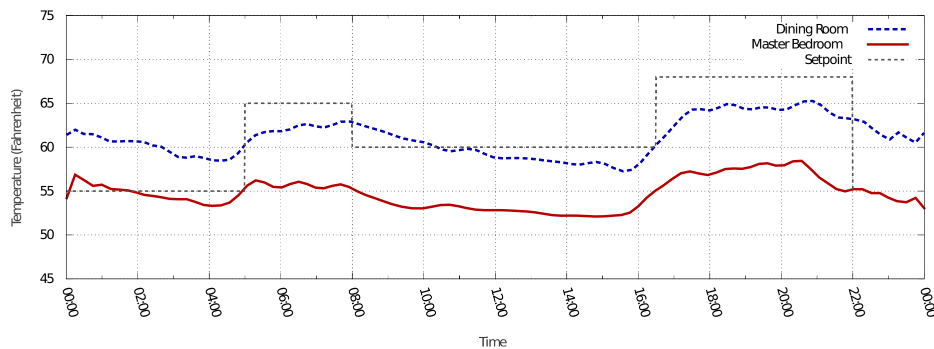

Figure 3.8: Observed room temperatures for two rooms on an average day in December 2013. The x-axis is the time of day, and the y-axis is the measured temperature. The master bedroom is located on the second floor and receives less air flow from the furnace, resulting in dramatic temperature reductions.

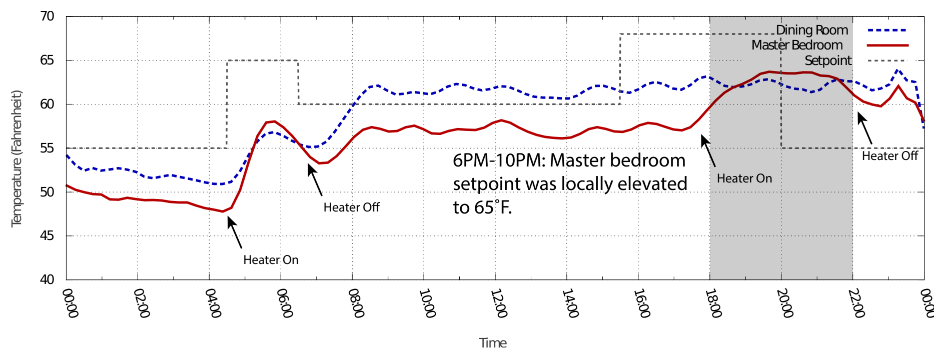

Figure 3.9: Observed room temperatures for the master bedroom and dining room after adding localized control to master bedroom. The x-axis is the time of day, and the y-axis is the measured temperature. A space heater is used to increase the master bedroom temperature from 6PM-10PM and from 4:30AM-6:30AM. For the rest of the day, the temperature in the master bedroom is allowed to drift away from its setpoint.

After adding localized control to target temperature variations on the second floor in Home C, the occupants were able to turn the global thermostat settings down by roughly 10°F during the second half of the month of December. Figure 3.8 shows the temperature in two rooms as a function of time before adding localized control. Clearly, the temperature in the second-story master bedroom is several degrees below the setpoint of the furnace's thermostat, making that room uncomfortable. Figure 3.9 shows a plot of the temperatures of the same two rooms as a function of time after adding localized control and setting back the thermostat. In this plot, the temperature in the master bedroom is higher during the hours between 6:00 PM and 10:00 PM. The invariant between Figures 3.8 and 3.9 is the temperature setpoint in the master bedroom between 6:00-10:00 PM.

We estimate that Home C used roughly 18% less natural gas after turning the thermostat down. Table 3.3 shows our estimates of the energy savings in Home C if the setback had been done for the entire month of December. The 20 therm reduction in natural gas consumption during the month of December in Home C is equivalent in terms of energy to 586 kWh of electric power — *more than 10 times the additional energy we expended to locally heat the master bedroom*[4].

To generate the projections shown in Table 3.3, we used a metric known as *heating degree days* (HDD) to estimate the natural gas used by the furnace. The number of heating degree days for some time interval is the number of degrees the furnace must heat the house above outside temperature multiplied by the length of the time interval in days. A heating degree day over some interval T is defined as

$$HDD = \int_T (T_{room}(t) - T_{ambient}(t))dt \qquad (3.1)$$

Where $T_{room}(t)$ is the room temperature defined by the thermostat's setpoint, and $T_{ambient}(t)$ is the outside air temperature[5]. For example, if

[4] 1 Therm = 100,000 BTU US, and 1 kWh = 3412 Btu, therefore 1 Therm ≈ 29.30 kWh and 20 Therms ≈ 586 kWh [17].

the outside temperature was 20 degrees cooler than the thermostat's setpoint for one day, we would count this as 20 heating degree days.

For each thermostat schedule shown in Table 3.2, we computed the number of heating degree days for the month of December using the integral in Equation 3.1. The number of heating degree days is different for the two schedules because the setpoint $T_{room}(t)$ is different, while $T_{ambient}(t)$ was the same for both schedules.

Our local utility (Madison Gas and Electric) models the energy usage of a furnace as a linear function of heating degree days [83]. Under this assumption, the natural gas usage of the furnace should also decrease by 18% because the furnace is the only gas-fired appliance in the house. This model gives a good first-order approximation of the energy usage of a heating or cooling system. By setting the thermostat back in Home C, we were able to achieve an 18% reduction in heating degree days during the month of December while maintaining thermostat schedule in the spaces used most by the residents.

Another important result of this work is that it incentivizes people to use less energy by providing additional benefit: increased comfort. Since the user's point of interaction with the system is the setpoint for the given room, logically users will maximize their comfort by defining a setpoint to suit their needs. Thus, our goal in this work is to optimize the residents' comfort by minimizing the temperature error in targeted areas of the house during time periods when we knew those areas would be occupied. In this way, we improve the efficiency of the HVAC system by conditioning the rooms in accordance with the user's desire, while reducing energy wasted by conditioning rooms that are not in use.

3.4 SUMMARY OF HOT, COLD, AND IN BETWEEN

Hot, Cold, and In Between is an IoT building automation system that monitors and controls the heating and cooling system within single family homes. We demonstrated that it can reduce the energy consumed by HVAC systems

[5] By this definition, it is possible to have a fractional number of HDD because the time interval T could be a fractional number of days.

while making spaces more comfortable for the building occupants. When used in conjunction with Emonix, our HVAC control system can reduce the overall building energy consumption by 15-20%. We next turn our attention to water treatment systems, which are an important contributor to groundwater pollution.

4 AWESOME & SPOCK: AN IOT SYSTEM TO IMPROVE THE EFFICIENCY OF WATER TREATMENT SYSTEMS

Building water usage is another important resource to consider as we construct a comprehensive platform to manage resource consumption. Water and energy consumption within a building are loosely coupled through occupancy. A large body of work in the smart buildings and home automation research community focuses on the problem of inferring building or room occupancy using sensor data.

In Madison and other metropolitan areas, water softeners are a primary source of Sodium and Chloride ion pollution [15, 47]. Pollution of fresh water sources is a major concern because it threatens the supply of potable water on which urban populations depend. Salt waste produced by softeners is discharged into waste water treatment facilities. Once dissolved, it is difficult and expensive to remove.

To combat the increasing concentration of pollutants being discharged by water treatment systems, we designed two complementary systems: AWESOME[1] and SPOCK[2]. The goal of this work is to increase the amount of water that a softener can treat between regenerations, thereby reducing the amount of salt per gallon required to treat hard water. We accomplish this by introducing an adaptive feedback control system that can monitor the volume and quality of the treated water and automatically make real-time decisions about when to regenerate the filtration medium. AWESOME is unique because it is a *bolt-on* solution that can be added to any existing water softener system to save salt. In contrast to other water-saving control systems, it relies very little on human intervention.

In SPOCK, we use raw historical sensor data collected from the water treatment system as input to a forecasting algorithm that predicts future sensor readings. We use forecast sensor readings rather than forecast occupancy

[1] AWESOME stands for **A** **W**at**E**r **S**oftener **O**nline Opti**M**ization **E**ngine
[2] SPOCK stands for **S**ensor **P**rediction and **O**nline **C**ontrol

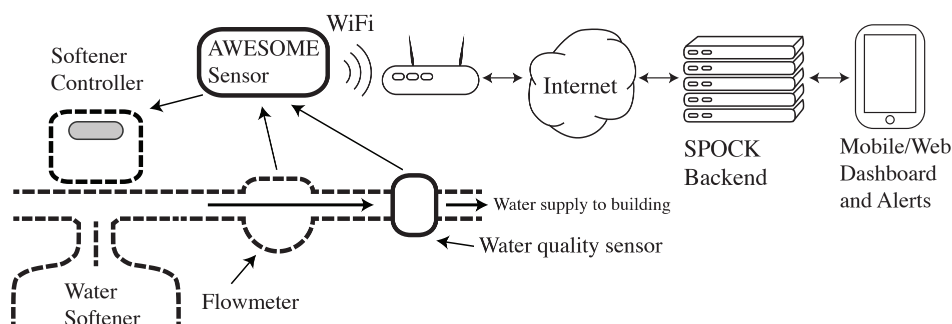

Figure 4.1: Dataflow diagram for the AWESOME system. Components with solid lines are part of AWESOME/SPOCK, and components with dashed lines are part of the existing softener system.

inferences to predict future resource-consuming activities within the building. We then make control decisions based directly on the forecast sensor readings.

The main challenge to building such a control system is that most commercially-available water quality sensors are delicate instruments whose output can drift over time. One key advantage of AWESOME is that it includes adaptive control algorithms that can tolerate sensor drift while detecting and responding correctly to true changes in water quality caused by depletion of the filtration medium.

What is a water softener? Water softeners stop lime buildup in pipes and equipment by removing dissolved minerals – Calcium and Magnesium ions – from tap water. This is typically accomplished by exchanging Calcium and Magnesium ions with Sodium ions. As Calcium and Magnesium-rich water passes through a filtration medium, Sodium ions, weakly bound to the medium, are released into solution. The Calcium and Magnesium ions replace the Sodium on the filtration medium. Eventually, the surface of the filtration medium becomes saturated with Calcium and Magnesium ions, and it can no longer treat water. It must be regenerated by flushing with concentrated salt brine, which replaces Calcium and Magnesium ions with Sodium, preparing the softener to treat more water.

However, large amounts of water pollution are not necessary to provide soft tap water in buildings. Anecdotally, we have observed that many softeners

are configured incorrectly, leading to excessive backwashing and wasteful salt consumption. Furthermore, there is a lack of information about water softener salt consumption that leads to malfunctioning systems.

Other types of water treatment systems, such as reverse osmosis and deionizers, also discharge pollutants. We chose to work with water softeners in this study because we have access to several large softener systems on campus. In principle, our techniques could be ported to work with various other treatment systems.

In our pilot deployments in three UW residence halls (Chadbourne, Sellery, and Leopold), we used AWESOME to initiate water softener regeneration based on water quality measurements. Chadbourne Hall is an eight-story residence hall that houses 600 students and a dining facility. Sellery Hall a six-story residence hall housing 1,200 students that does not include on-site dining. Leopold Hall is a four-story residence hall housing 200 students. As a result, *we reduced the salt used by an average of 27%*.

The AWESOME System and How it Addresses Inefficiencies

Most water treatment systems use open-loop control, meaning that there is no sensor on the outgoing treated water to inform decisions about when to regenerate the filtration medium. Instead, the control systems usually use simple time-based (regenerate once a week on Tuesday) or flow-based (regenerate every 1000 gallons) schedules.

Inconsistencies in the incoming water quality, degradation of the filtration medium, or variation of the water usage pattern of the building can make the system unstable. In Madison, municipal water is drawn from a pool of 17 wells, each of which supplies water with different hardness. The variation ranges from 20 to 30 grains per gallon[3] [47].

Water softeners tend to be set to use more salt than strictly required for several reasons. Adding too much salt doesn't negatively impact the piping of the building and won't be noticed by the user. The cost of maintenance,

[3] A *grain* is a unit of mass, approximately equal to 65 mg. Water hardness, the concentration of Calcium and Magnesium ions, is conventionally measured in grains per gallon.

especially in large buildings, for cleaning up lime is higher than the cost of additional bags of salt. This is primarily due to the cost of the labor involved. However, gross overusage of salt, as we have found to be common among the buildings we've studied, is also expensive in the long run.

Because water softener systems are typically located in remote service areas, problems can go undetected for months or years. Furthermore, existing softener controllers do not typically display enough information for maintenance staff to detect misconfigurations.

Misprogrammed controllers can cause inefficiencies in the softener system by allowing either too much or too little water to flow through the resin bed before it is depleted. Regenerating the resin bed too early can result in excessive salt use and increased Sodium Chloride pollution. Regenerating too late can cause the resin bed to become totally depleted, resulting in hard water being distributed to the building. This can cause pipes to clog and eventually destroy equipment. Unfortunately, many existing softener controllers are difficult to program. Even experienced maintenance personnel can make mistakes in programming the softener controllers, resulting in over-softened or under-softened water. Reprogramming softeners is routinely required after a power failure or after a changeover from daylight savings time.

Low flow rates through a softener tank can result in nonuniform flow of water through the filtration medium, causing some regions to deplete more quickly than others. Under low-flow conditions, water tends to move through a column in the center of the softener tank, quickly depleting the medium in that region. For this reason, it may be necessary to regenerate the water softener more frequently during periods of low consumption.

Variability of incoming water quality creates obstacles when provisioning a treatment system because it is difficult to predict the amount of water a system can treat before it needs to be regenerated. Since the hardness of incoming water commonly varies by 10% or more over the course of days or weeks, softeners are often configured to deal with the worst case hardness. Even when incoming water has relatively low hardness, the softener system will still be regenerated on the same schedule as under maximum hardness conditions because stock water softeners do not have any way of sensing water quality.

AWESOME System and its Advantages When a water softener regenerates, it flushes its filtration medium with salt and water, which must be sent down the drain. The more frequently it regenerates, the more water and salt the system uses. Existing water softener control systems perform poorly because they cannot adapt their regeneration schedules to changes in water quality and usage. Since stock controllers are typically configured to regenerate more frequently than necessary to maintain soft water, their consumption of salt and water is wasteful.

Our goal in developing AWESOME was to design a system which will regenerate the softener tank only when strictly necessary. In so doing, the softener can treat more water between regenerations, which conserves resources.

By sensing the water after it has been treated by the water softener, AWESOME determines when it is appropriate to regenerate. However, we found that it is difficult to distinguish hard water from soft water using data gathered from just one sensor because water quality sensors have a tendency to drift over time. Instead, our backend ties together noisy sensor data using domain knowledge to generate a reliable regeneration output.

Key Contributions

The work described in this paper covers both significant research and engineering problems, making the following contributions:

- **We develop an adaptive control algorithm to predict when the softeners need to be regenerated based on our sensor inputs.**
- **We design a closed-loop system to measure, record, and respond to water quality sensor inputs.**
- **We deploy AWESOME in three locations over the course of two years.**
- **We develop a sensor value forecasting algorithm that predicts future sensor readings based on historical data.**

- **We evaluate our algorithm using data collected from our deployments, demonstrating that our methods can reduce the salt used by 15-45%.**

4.1 BACKGROUND

The main function of a water softener is to protect water heaters and fixtures from lime buildup. When hard water is heated, dissolved Calcium becomes insoluble and forms scale deposits on pipes and fixtures, requiring expensive and inconvenient repairs. Particularly in large commercial and industrial operations where hot water or ultrapure water has a mission-critical function, water softeners are often an indispensable component of the water supply chain.

The main working component of a water softener is a large tank filled with tiny plastic beads. The beads are made of a specialized plastic resin that is chemically engineered to attract metal ions—in particular, Calcium (Ca^{2+}), Magnesium (Mg^{2+}), and Sodium (Na^{+}). The beads are initially coated with Na^{+} ions, as shown in Figure 4.3 (a). As hard, Calcium-containing water flows past the resin beads, the Calcium ions adhere to the surface of the beads, releasing Sodium ions into solution, as shown in Figure 4.3 (b) and (c). This Sodium-rich *soft water* supplies boilers, deionizers, and other sensitive devices in the building. Soft water is safe for equipment because Sodium ions have far less propensity to precipitate out of solution than Calcium and Magnesium ions, meaning that soft water will not leave behind scale on the equipment it contacts.

After the water softener has treated a large volume of hard water, its resin beads are fully saturated with Calcium ions, as shown in Figure 4.3 (d). Once this happens, the resin must be *regenerated* before it can treat more water. To regenerate the softener, the resin medium is soaked in a concentrated solution of table salt (NaCl) and water. Sodium ions in the salt brine replace Calcium ions on the surface of the resin beads, returning the filtration medium to the diagram shown in Figure 4.3 (a).

A controller, present on all water softeners, is responsible for deciding when the filtration medium needs to be regenerated. However, even most modern controllers do not use water quality sensors to determine when the resin needs

to be regenerated. Instead, they track the total water that has flowed through the filtration medium since the last regeneration, and trigger a regeneration after the total flow has reached a pre-defined level. We believe that this is because most water quality sensors are expensive and produce noisy data which is not easily interpretable by a simple controller. Instead, controllers guess at when the filtration medium needs to be regenerated. If the controller regenerates too late, it will allow the resin to deplete, sending hard water to the building, which could be damaging. If it regenerates too early, it will use more salt than necessary, but it will not destroy any building components.

To err on the side of caution, most controllers regenerate the filtration medium long before it is actually depleted in order to avoid supplying hard water to the building, which could potentially require costly repairs. Since the same amount of salt is needed to regenerate a softener that is 80% or 99% depleted, it is wasteful to regenerate the filtration medium early.

Standard Regen Method: Reserve Capacity

Most water softeners currently use a method called *reserve capacity* to schedule regenerations. The system's reserve capacity is the expected volume of water that will be used by the building in a 24-hour period. This estimate is loosely based on the building size, number of occupants, occupant activities, etc., and it is hard-coded into the water softener's controller at installation time by the installer. The water softener is then put into service. It begins treating water, and, using a flowmeter, it subtracts the volume of water it treats from its theoretical capacity. Each morning at 2 AM (a time when water usage is assumed to be low), the water softener controller compares the remaining theoretical capacity to the reserve capacity to determine if it has enough remaining capacity to treat water for another full day. If the softener's remaining capacity is less than the reserve capacity, it regenerates at 2 AM—even if the softener's remaining capacity is one gallon less than the reserve. If the remaining capacity is greater than the reserve capacity, it waits until the next day at 2 AM to recompute.

Clearly, there are many potential pitfalls in this approach. The building's actual water consumption patterns may not match the reserve capacity hard-

coded into the softener's controller. Also, 2 AM may not be the only opportune time of day to regenerate: there may be multiple periods during the course of a day when flow is low enough to regenerate.

To study the efficiency of the reserve capacity method, we took the flow rate measurements gathered from one of our installations and processed them with the reserve capacity algorithm. The algorithm identified times when the water softener would have regenerated using a reserve capacity. We varied the reserve capacity—the expected volume of water used by a building in a day—from 2,000 to 26,000 gallons, and we recorded the actual number of gallons at which the softener regenerated. This experiment was done on a dataset consisting of 225 days of flow rate data starting in June, 2015. The results of this test are shown in Figure 4.2.

The average water usage of the building that this test was run in is consistently about 14,000 gallons. However, to ensure that we never overrun the softener's theoretical capacity of 36,000 gallons, the reserve capacity needs to be set to at least 18,000 gallons. If we set the reserve capacity to 18k gallons, the softener regenerates on average at 30k gallons, wasting 17% of its theoretical capacity on average. That is the best the reserve capacity algorithm can achieve using rigorous statistical analysis of the building's water consumption with nine month's data as input—methods and data that are not available to water softener installers.

The problem that this work tries to solve is to use the building's actual water consumption patterns to adaptively choose a regeneration time that will maximize the system's efficiency without regenerating too late.

4.2 DESCRIPTION OF AWESOME

The AWESOME system consists of custom hardware and software components that measure water quality, process the recorded data, and respond.

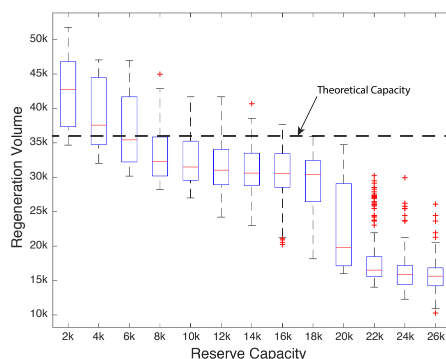

Figure 4.2: Reserve capacity (x-axis) vs. regeneration volume (y-axis) for flow data collected in a senior living facility over the course of 225 days in 2015-2016. For the same flow data, we simulated reserve capacity-based regenerations, using the same theoretical water softener tank capacity of 36k gallons. We varied the reserve capacity from 2k to 26k gallons to determine what the optimal setting should be. To ensure that we do not overrun the theoretical capacity of the softener, we must set the reserve capacity to 18k gallons, which wastes 6k gallons per cycle on average—more than 17% of the theoretical capacity.

Measuring Water Quality With AWESOME

AWESOME uses water flow and quality sensors to track the status of a water softener in real time. We constructed custom hardware (shown in **??**) to gather and record data from water softeners. The data it collects is transmitted to a remote database for storage and post-processing (described in section 4.2). When backend algorithms running on the remote database server determine that the water softener needs to be regenerated, they transmit a signal to AWESOME, which forces the water softener to regenerate. A diagram of the dataflow used by AWESOME is shown in Figure 4.1.

AWESOME uses several sensors to track the health of a water softener system:

A Calcium Ion Selective Electrode (ISE) produces a voltage signal that is proportional to the Calcium concentration of the water sample. These devices are extremely sensitive to temperature and other factors, and their outputs are known to drift over time. By itself, the Calcium ISE cannot tell us the

concentration of Calcium in our water samples, so we need to use auxiliary sensors to augment our data. Manufacturers of Calcium ISEs warn that their products should not be held in test solutions for long periods of time. This concerned us because our approach requires that we use ISEs to continuously gather water quality samples, with the probes themselves constantly submerged in water samples. We expected to have to replace the sensors after a few months of data collection since we were not using the ISEs according to the manufacturer's recommendations. However, we have had several deployments of AWESOME running continuously for two years, and we have not had an failures in the ISEs.

In section 4.1, we mentioned that both Calcium and Magnesium ions contribute to total water hardness. Both are always present in hard water, though their ratios can vary depending on the source of the water. Our sensor system measures only Calcium ion concentration, since it is an indicator of total water hardness. Water that has a significant concentration of Calcium ions will also have a significant concentration of Magnesium ions.

A Water Temperature Sensor is used in combination with the Calcium ISE to estimate the Calcium ion concentration in the water sample. We use the Nernst Equation[59] to combine the Calcium ISE voltage and the water temperature to get an estimate of the Calcium ion concentration. Because the Calcium ISE's output voltage is prone to drift over time, we cannot use the raw output of the Nernst equation to make decisions about whether or not the water is hard.

A Flowmeter is already installed in most commercial water softeners. Flow rate data is used by the stock water softener controller to decide when the softener should regenerate—in fact, this is the only sensor included by default on most commercial softener systems. Fortunately, all water softener flow meters use the same interface to communicate with their controllers. AWESOME can intercept and record the flow rate signal.

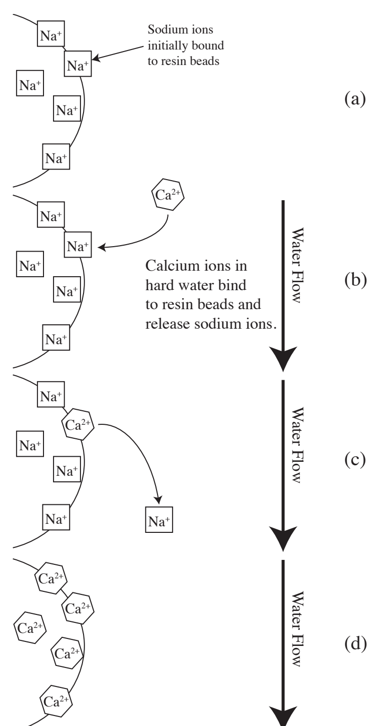

Figure 4.3: A diagrammatic overview of the way a water softener functions, explained in section 4.1.

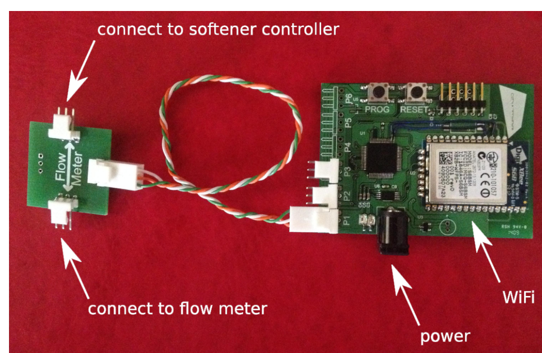

Figure 4.4: Photograph of an AWESOME board.

Backend Data Processing in AWESOME

We now discuss several approaches to processing the data gathered by our sensors. The goal of our data processing algorithms is to correctly and quickly identify filtration medium depletion. As we mentioned in section 4.1, the main challenge we face is that our water quality sensors produce noisy data. In particular, the output of our sensors may drift over time, and variability in flow may result in errant spikes in the sensor output, even though the softener is not actually producing hard water.

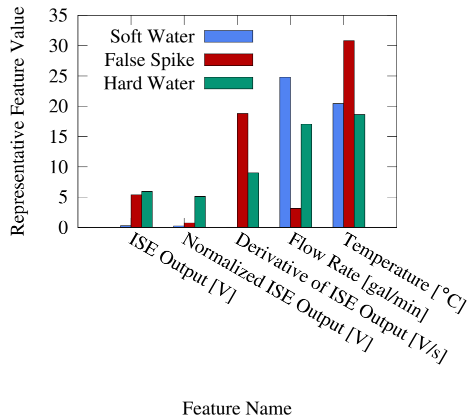

Figure 4.5: A demonstration of how the features used by the SVM differ for soft water, truly hard water, and false spikes (noise).

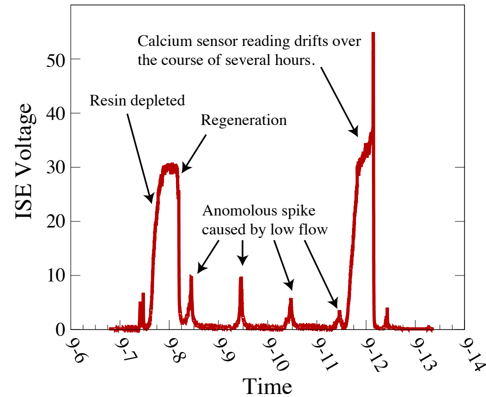

Figure 4.6: Example trace of some raw hardness data gathered from the Calcium ISE. At night, the flow through the water softener drops, creating anomalous spikes that could be confused for hard water output. On 9-12, the sensor output drifts over the course of several hours, even though the water hardness remains constant.

Feature	**Formula**
ISE Output	$[Ca^{2+}] = e^{k(V-a)}$
Flow Rate	$f[gal/minute]$
Temperature	$T[°C]$
Derivative of ISE Output	$\frac{d[Ca^{2+}]}{dt}$
Normalized ISE Output	$[Ca^{2+}]\left(1 - e^{-kf/T}\right)$

Table 4.1: Features used by our adaptive control algorithm to determine whether the softener system needs to be regenerated.

Figure 4.6 shows an example output trace of the data gathered by our hardness sensor before it has been processed by our backend algorithms, highlighting some sources of noise.

- **Low flow rates**, particularly during the night, confuse the sensor because stagnant water in the pipes heats up, disrupting our readings.

- **Drifting voltage output of our Calcium ion selective electrode** caused by degradation of the ion-selective membrane and the reference electrode can generate inaccurate readings.

The goal of our data processing algorithm is to identify true hard water samples correctly and reject anomalous readings. We fed the data gathered from our three pilot deployments into two learning algorithms (SVM and thresholding, described below), and evaluated the quality of predictions made by each algorithm.

Ground truth was gathered manually in each building using a chemical hardness test kit [58]. The test kit we used provides a reliable measurement of water hardness, but it is a time-consuming manual process which is not practical to automate.

The algorithms in this section were trained and tested by disabling regenerations on the water softeners in the pilot buildings and allowing their filtration media to deplete. Once the filtration media depleted, the outgoing water became hard, as shown in Figure 4.6. We allowed our sensors to collect data during the process of depletion, and we independently verified that the softener produced hard water by using a manual water hardness test kit [58]. Because we did not want to damage the buildings' hot water heaters, we only gathered data for a few hardness events in each building. For this reason, we have at most seven hard water events on which to train and test our algorithms.

Following the data collection, we disassembled one building's hot water heater to find out if the hard water events caused any damage. The building maintenance staff said there did not appear to be excessive lime buildup on the heating coils, leading us to believe that our experiments did not have a detrimental effect on the building's water system.

Why Simple Thresholding Does Not Work Well In the simplest data processing approach, we used a threshold on the ISE voltage to determine whether

the output water was hard. Above some user-defined output of the Calcium ISE, the water was considered hard, and a regeneration was initiated.

An advantage of this technique is that its performance is intuitive and predictable—an regeneration will always be initiated when the ISE voltage rises above the threshold, which can be made as low as desired. However, thresholding is not good at rejecting noise and drift in the ISE voltage output. Under low flow conditions, we frequently see spikes (usually at night, depicted in Figure 4.6) which the thresholding technique may interpret as medium depletion and erroneously regenerate the softener, wasting salt. Alternatively, we could increase the threshold above the maximum height that we expect for one of these erroneous spikes. However, this approach is also suboptimal because it does not regenerate the softener until the water has become very hard, which could damage pipes and equipment. We need a better approach which can accurately detect filtration medium depletion early, before the softener starts producing extremely hard water.

SVM-Based Technique Using an SVM, we can integrate readings from multiple sensors to detect hard water early. SVM takes as input a set of labeled training vectors (sensor readings labeled with ground truth). The underlying SVM algorithm [52] finds a linear function separating feature vectors in the two classes—hard water sample vectors lie on one side of the function, and soft water vectors lie on the other. Unknown feature vectors are classified based on which side of the linear function they fall.

The features we used as inputs to our learning algorithms are enumerated in Table 4.1. The first three features—hardness[4], flow rate, and water temperature—are measured directly from sensors. The last two, computed as functions of the first three, were added to reject noise and drift of the Calcium ISE. Figure 4.5 shows how the features we chose for the SVM vary depending on whether the sample water is soft, hard, or a false spike.

The ISE output is a function of the water's Calcium ion concentration, and it should generally be low for soft water and high for hard water. Since we want

[4]More specifically, the Calcium ion concentration $[Ca^{2+}]$ is computed as a function of the voltage output of the Calcium ISE (V).

to detect hard water before the Calcium ion concentration (ISE voltage) gets too high, we need to add more feature vectors to distinguish between false spikes and hard water.

To distinguish between hard water and false spikes, we also measure flow rate. False spikes generally occur because of low flow rate through the water softener. In addition, increased water temperature has a tendency to cause erroneous high ISE readings.

Perhaps the most important characteristic we studied is the shape of the ISE curve as it increases. In Figure 4.6, it is visually evident that the shape of a false spike is different from the shape of a true hardness event. We can take advantage of the difference in order to detect water hardness events early because we can identify the shape of the curve before we can identify its maximum amplitude.

We used the derivative of the Calcium ISE voltage output trace to measure shape. In Figure 4.5, it is clear that hard water events have a smaller derivative than false spike events. This is true almost universally in the dataset we collected from our pilot deployments. The reason for this is that at the end of a filtration cycle when the medium is almost depleted, the medium goes through an intermediate phase in which it slowly becomes less efficient at removing Calcium ions. That intermediate phase generally lasts for several hours. Erroneous spikes, on the other hand, are caused by changes in water use patterns which generally happen over much shorter time intervals.

By tracking multiple sensor inputs as well as the shape of the ISE output, we can much more accurately distinguish between truly hard water and false sensor spikes. In the next section, we evaluate our SVM-based technique, comparing it to simple thresholding. We demonstrate that we can detect and respond to resin depletion before water gets too hard to threaten building systems like hot water heaters and faucets.

4.3 SPOCK FORECASTING ALGORITHM

The AWESOME SVM-based technique works well for multi-tank softener systems in which a second backup unit can treat water while the first regenerates. But smaller water softeners do not have backup units available. Instead, they

bypass softener during regeneration, causing hard water to flow to the building. The goal of SPOCK is to choose a regeneration time that is likely to have low water use in order to bypass as little hard water as possible.

The algorithm we describe in this section takes flowrate, hardness, and other sensor readings (described above) and identifies an optimal time to regenerate the water softener based on predictions of future sensor readings. The data processing techniques described in this work can be broken down into three major steps:

1. Using historical data gathered from the AWESOME sensor, forecast water flow and hardness for a 24-hour time horizon.

2. Construct two cost functions from the forecast data: one that represents the cost of regenerating the water softener (increasing during times of higher flow) and one that represents the cost of not regenerating (increasing with higher hardness). Regenerating the water softener during times when water flow is high would result in large volumes of untreated hard water being passed through building systems. This cost function will have large values during times when flow is forecast to be high and small values during times when flow is forecast to be low.

3. Compare the two cost functions to find the optimal time window for regeneration, trading water usage efficiency with utility.

Sensor Forecasting

The two metrics we are interested in forecasting are water flow and water hardness. These are the two important inputs to the cost function, which we will use to decide when to regenerate the water softener.

Flow Forecasting We want to regenerate the water softener at a time when minimal water will be used by the building. To choose such a time, we will forecast water flow for a 24 hour time horizon. We do this because we do not want to begin regenerating the water softener at a time when the building's

water consumption is likely to increase significantly in the near future. Since the water softener takes 60-90 minutes to regenerate (during which time it cannot treat water), we need to find a time frame when water flow is likely to be minimal.

A statistical analysis of water flow patterns in various buildings we studied indicate that the flowrate function is not a stationary process[5]. This finding makes sense because activities in a building will likely follow daily circadian fluctuations. A histogram of water flow rates from a residence hall is shown in Figure 4.8. At 4 AM, when most residents are asleep, the histogram indicates that the recorded flow rates are mostly zero. Flow rates increase around 8 AM when residents wake up to go to class. At noon, flowrates increase further because lunch is served in the building's cafeteria. Since the distribution of water flow rates changes during the course of the day, we conclude that the process is nonstationary.

Since the flowrate signal is nonstationary, we need an analysis method that can deal with data that has a distribution that varies slowly over time. We chose to use an autoregressive model as a tool to forecast future flowrate patterns [87].

The flowrate signal that we are studying tends to be bursty—it tends to be high for a short time when a fixture is turned on and low when the fixture is off. In a large building such as those that we study here, that burstyness is contributed by hundreds of fixtures, each contributing to a noisy signal with a lot of high-frequency components. Unfortunately for us, autoregressive models are fairly sensitive to noise in the input signal. To deal with noise in the flowrate signal, we chose to first pass it through an integrator, which computes the total volume of water consumed by the building since the beginning of the timeframe of interest. This cumulative flow signal is probably a more useful metric anyway, since we are often concerned with the total amount of resource consumption.

The cumulative volume signal, having been passed through an integrator, is very smooth, and is a good candidate for use with an autoregressive model. The autoregressive model then predicts the cumulative flow used by the building on

[5]A stationary process is one whose statistical properties do not change as a function of time. In other words, X_t is stationary iff $F_X(x_{t_1}...x_{t_k}) = F_X(x_{t_{1+l}}...x_{t_{k+l}})$

a minute-by-minute basis, 24 hours into the future. The results of the predictions made by the autoregressive model will be presented in Section 4.4.

The minute-by-minute data, while useful to the autoregressive model because of its high information content, is not required for the purposes of constructing the cost function. For the water softener application, we do not need to know the exact moment when it is optimal to regenerate. Instead, we are looking for a time window—typically on the order of several hours in length—when a regeneration would be appropriate. For this reason, we downsample the cumulative flow predictions from 1-minute sampling frequency to 1-hour sampling frequency. The downsampling has the effect of lowpass filtering the flow forecasts.

The final step in our forecasting process is to differentiate the downsampled cumulative volume predictions. This gives us the predicted *flow rates* on an hour-by-hour basis for a 24-hour time horizon. In the next steps of our algorithm, we will search this forecast for time periods when flow is minimal—times when it will be least costly to bypass the water softener and regenerate.

Hardness Forecasting In addition to forecasting flow, we also need to predict water hardness readings, since these will inform our decision to regenerate. If we do not predict that the water softener's filtration medium will be depleted in the near future (i.e. water will not be hard), there is no point in regenerating.

Our hardness forecasting technique, like our flow forecasting technique, uses an autoregressive model to predict future sensor input values. Unlike the flow rate signal, the hardness signal is sparse. Most of the hardness readings are zero or nearly zero—this is expected because when the water softener is working properly, it removes all hardness from the water. It is only when the water softener's filtration medium depletes that we get nonzero hardness readings. The sparse nature of the hardness readings and the inherent nonlinearity of the hardness with respect to the cumulative water flow through the softener poses a challenge for learning linear models, such as, autoregressive models.

To account for the inherent nonlinearity so described, we use two operating point models (for low/high cumulative flow since the last regeneration). This is justified on the basis that the nonlinearity of the hardness measurement only

manifests itself in the high flow region. A parallel can be drawn between our approach here and the general practice of linearizing nonlinear systems near typical operating points for the design of closed-loop controllers.

Additionally, since the hardness measurements depend on the cumulative flow through the softener since the last regeneration, in addition to the history of the hardness measurements themselves, the model we learn can be expressed in the following discrete-time state space notation ($\mathtt{flow}(\dots)$ refers to the cumulative flow since the last regeneration):

$$X(k+1) = \begin{bmatrix} A_{\text{flow-flow}} & 0_{\text{flow-hardness}} \\ C_{\text{hardness-flow}} & D_{\text{hardness-hardness}} \end{bmatrix} X(k)$$

$$X(k) = \begin{bmatrix} \text{flow}(k) \\ \dots \\ \text{flow}(k-20) \\ \text{hardness}(k) \\ \dots \\ \text{hardness}(k-20) \end{bmatrix}$$

$A_{\text{flow-flow}}$ captures the nature of the water usage by the residents of the building. Since this quantification is subject to rapid, unpredictable changes, we re-learn the flow forecasting model after each regeneration cycle thereby using the most recent water usage patterns for the forecasting, eliminating forecasting bias due to historical data.

The dependence of hardness measurements on their own history and the flow is quantified by the coefficient matrices of the mode $C_{\text{hardness-flow}}$ and $D_{\text{hardness-hardness}}$.

The model so described is learned from recorded data for two operating regions (low/high cumulative flow since the last regeneration). When the actual forecasting is done, one of the models is chosen based on the flow since the last regeneration at the time of forecasting.

Cost Function Setup At the most basic level, there are really only two actions a water softener controller can take at any moment: regenerate the water softener now or wait until later to regenerate the water softener. After we have generated predictions of the flow and hardness signals for the next 24-hour timeframe, we use them to generate cost functions that capture the costs of either (1) regenerating the water softener or (2) not regenerating the water softener[6].

The cost of regenerating a water softener is largely measured in terms of the salt and water consumed in a regeneration. We know, however, that we will have to regenerate the softener occasionally, and our objective is to reduce the number of regenerations per gallon of water treated. The cost of not regenerating the water softener is captured by the potential damage to building infrastructure caused by flowing hard water to the building after the filtration medium has depleted. For this reason, we will use

$$C_{Regeneration} = af(t)$$

$$C_{NotRegenerating} = bf(t)H(t)$$

where a and b are normalizing constants, $f(t)$ is our forecast of the instantaneous flow rate through the softener, and $H(t)$ is our forecast of the hardness of the water after being treated by the water softener. These two cost functions are generated from our forecasts, which are based on historical sensor readings. Normalizing constants a and b are introduced to make $C_{Regeneration}(t)$ and $C_{NotRegenerating}(t)$ similar in magnitude. They can be used as knobs to tune the system to be more conservative or more aggressive at saving resources.

Cost Function Processing To determine when to regenerate the softener, we compare the cost functions $C_{Regeneration}$ and $C_{NotRegenerating}$

$$C_{diff} = C_{Regeneration} - C_{NotRegenerating}$$

[6]Here we use the term *cost* to mean not the monetary cost of taking one action or another, but the strain on infrastructure and resources.

C_{diff} **Range**	Indicative Of
Large +ve	High Flow, Low Hardness
≈ 0	Low Flow, Low Hardness
Large -ve	High Flow, High Hardness

Table 4.2: Ranges of the cost function C_{diff}, and the corresponding sensor readings that cause them.

An example trace of C_{diff} is shown in Figure 4.7 in the bottom graph. Corresponding measurements of hardness and flow are shown in the plot on top. Ranges of C_{diff} and their corresponding meanings in terms of water hardness and flow are given in Table 4.2.

Our goal is to regenerate the softener before the hardness gets too high. However, we want to schedule the regeneration cycle during a period of low flow, because we will need to bypass the water softener during the regeneration cycle, and we do not want to flow a large volume of untreated water to the building.

Our algorithm for choosing a regeneration time based on a computed cost function is as follows:

1. Find minima of C_{diff}, that satisfy $C_{diff}(t_{min}) < T$, where T is a prechosen threshold. This represents a time when the hardness has increased so that the cost of not regenerating is higher than the cost of regenerating (see Figure 4.7 (*)).

2. Search the cost function $C_{diff}(t)$ for values of $t < t_{min}$ that satisfy $C_{diff} \approx 0$ and $\frac{dC_{diff}}{dt} \approx 0$. This is the optimal time to regenerate.

If, when processing $C_{diff}(t)$, we find a minimum that is significantly less than zero, we know that our forecasts indicate that the water will become hard at the time the minimum occurs. If the threshold T is too close to zero, this algorithm may initiate a regeneration too early, perhaps in response to a spurious increase in hardness readings which results in elevated hardness forecasts. For this evaluation, we chose T by trial and error. Our algorithm is not highly sensitive to the choice of T, but we have observed that setting it too

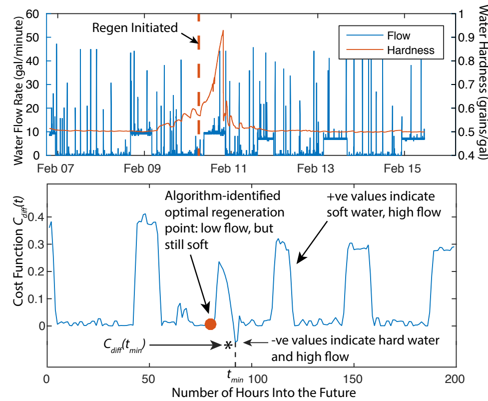

Figure 4.7: Top: Water flow and hardness measurements (y-axis) as a function of time (x-axis). Bottom: our cost function (y-axis) as a function of time (x-axis). Large positive values of the cost function indicate high flow & low hardness.

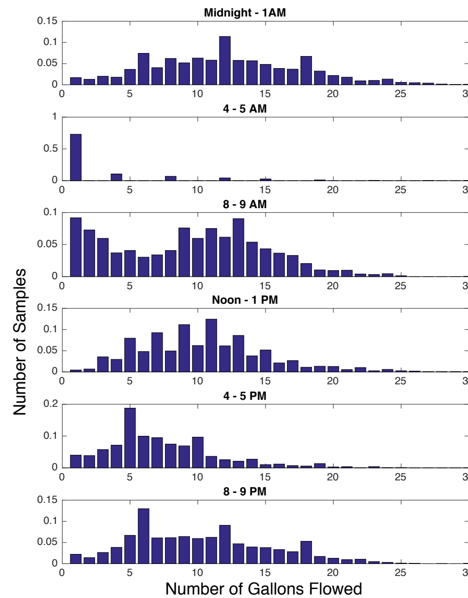

Figure 4.8: A histogram of flow rate measurements taken in a residential building over several months in 2015. The distribution of water flow rate changes depending on hour of the day, so the flow rate function is not stationary.

close to zero can cause early regenerations, which would reduce efficiency. We are calling that time t_{min}. However, by the time that minimum occurs, it will be too late to regenerate—we should have regenerated before the water became hard, or, in terms of the cost function, before C_{diff} became negative.

We want to schedule a regeneration when C_{diff} is close to zero. According to the way we've defined our cost function, values near zero indicate that we have low flow and low hardness. To find such a point, we will trace our cost function back from t_{min} to find a point on the cost function where C_{diff} is close to zero just before it becomes negative.

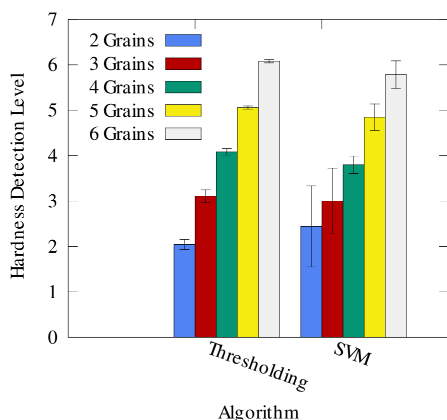

Figure 4.9: The hardness at which each algorithm detected filtration medium depletion. We show the detection level for each of five different training levels. Our goal is to detect medium depletion at the lowest possible hardness level while minimizing the number of false positive identifications

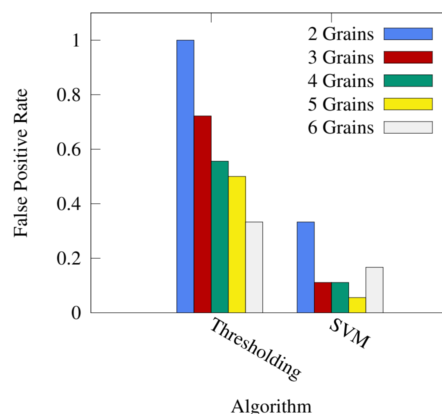

Figure 4.10: False positive rates for the learning algorithms evaluated. Each algorithm was tested with five different training thresholds. In general, lower training thresholds result in more false positives.

4.4 EVALUATION

Here we evaluate SPOCK and AWESOME algorithms.

AWESOME

In this section, we test our learning algorithms for robustness, showing that they can correctly identify filtration medium depletion in a water softener. We go on to discuss the results of pilot deployments of AWESOME, showing that it can reduce salt consumption of water softeners by 15-45% compared to stock water softener controllers.

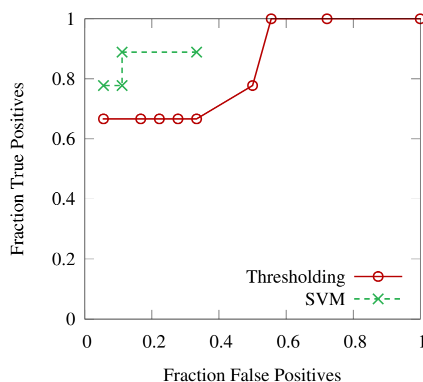

Figure 4.11: Receiver operating curve for the learning algorithms we analyzed. Algorithms that perform well will have ROCs that are close to the upper left corner of the graph. In that region, the algorithm generates a low fraction of false positives and a large fraction of true positives.

Learning Algorithm Evaluation

To evaluate our learning algorithms, we feed data gathered from our pilot deployments into each of several candidate algorithms and test their performance. Our goal is to find out

1. How quickly can each algorithm identify depletion of the filtration medium?

2. How resistant is each algorithm to anomalous sensor inputs (noise)?

To identify filtration medium depletion, we will necessarily have to allow the medium to deplete and start producing hard water. Our goal is to minimize the amount of hard water that needs to be produced before our algorithm can identify that the medium is depleted. If we supply too much hard water to a building before initiating a regeneration, we could damage the water heater and other building systems.

SVM is a supervised learning algorithms, meaning that it requires a training phase in which input data labeled with ground truth is fed to the algorithm. When constructing the training sets for SVM, we need to label each feature

vector with ground truth. This labeling process is subjective because the water hardness—concentration of Calcium and Magnesium ions—is a real number which must be converted to a binary value (0 or 1). So to label our training data, we must apply a threshold, which we call a **training threshold**, below which water is considered to be soft (0), and above which it is considered to be hard (1).

The threshold we apply will affect our algorithms' ability to distinguish between sets. If the threshold is too low, the algorithm may classify all unknown inputs as hard. If it is too high, we may miss all hardness events. Our goal in AWESOME is to produce training sets that will identify filtration medium depletion early (before the outgoing water has become too hard) while rejecting noise. In the parlance of hypothesis testing, we wish to optimize the ratio of true positives to false positives.

The lower we set the training threshold, the earlier our algorithms will in general be able to detect filtration medium depletion. Figure 4.9 shows a plot of the average outgoing water hardness level at which each algorithm can detect medium depletion for several different training thresholds. The x-axis lists the algorithms we studied, and the y-axis gives the average water hardness required for each algorithm to identify a filtration medium depletion. To be practical, the detection level should be below five grains per gallon because allowing the hardness to get any higher would damage building systems.

In our evaluation, we trained and tested each algorithm to five different training thresholds (2-6 grains hardness). As one would expect, the higher the training threshold, the higher the output hardness needs to be for an algorithm to identify filtration medium depletion.

However, at low training thresholds, the learning algorithms tend to produce more false positives. Figure 4.10 shows the false positive rates on the y-axis with the algorithms we tested on the x-axis.

To strike a balance between minimizing false positives and minimizing the hardness detection level, we find that it is best to train the learning algorithms to identify depletion at three grains hardness.

The receiver operating curve (ROC) shown in Figure 4.17 shows the tradeoff between false positives and true positives for SVM and thresholding. On the

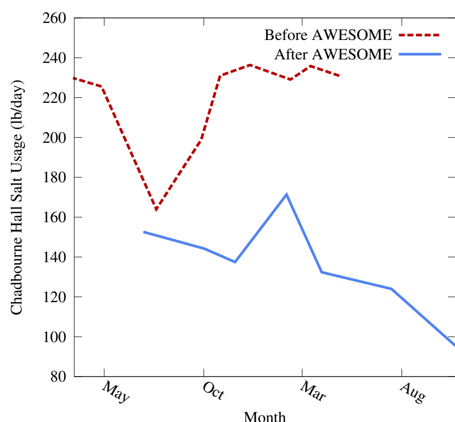

Figure 4.12: Salt usage in Chadbourne Hall over a two-year timespan. Salt consumption declined dramatically following the installation of AWESOME (solid blue line). This data is based on salt deliveries made by Kreger Salt Sales, the supplier of softener salt for Chadbourne Hall.

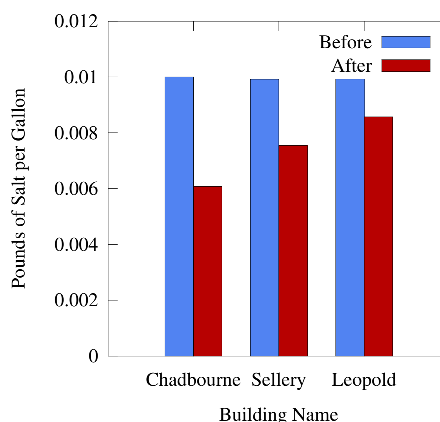

Figure 4.13: Amount of salt used per gallon of water treated in each of the three buildings where we deployed AWESOME.

y-axis, true positives are plotted as a function of false positives. These are parametric curves in which different hardness thresholds are used to train each control algorithm. Ideally, we would like our algorithms to operate as close to the upper left corner of the graph as possible: we want to increase the percentage of true positives as much as possible while keeping the fraction of false positives as low as possible. In the graph, SVM performs much better than thresholding because we only need to increase the proportion of false positives to 30% in order to achieve 90% true positive detection

Cost Savings

AWESOME was installed in Chadbourne Hall on the UW campus in July 2013, replacing the stock softener controller. Using our adaptive control algorithms

[7] These computations are based on orders of pallets containing 49 $\times$ 50-lb bags of salt, costing \$7.64 per bag.

Building	Year 1	Year 2	Savings
Chadbourne	68,600 lb ($10,482)	46,550 lb ($7,113)	22,050 lb ($3369)
Sellery	41,445 lb ($6,333)	31,500 lb ($4813)	9,945 lb ($1520)
Leopold	17,000 lb ($2,597)	14,700 lb ($2,246)	2,300 lb ($351)
	127,045 lb ($19412)	92,750 lb ($14172)	**34,295 lb ($5,240) 27% saved**

Table 4.3: Comparison of salt consumed in the three pilot buildings before and after installation of AWESOME [7].

to control the water softener, we were able to save $4118 in salt purchases over a six-month period ending January 2014. Table 4.3 shows the salt savings achieved after installing AWESOME.

For the pilot deployments discussed in this section, we used a thresholding algorithm to detect and respond to filtration medium depletion. To reduce the number of unnecessary regenerations caused by false positives, we used a threshold of ten grains per gallon. A downside of this approach is that it does not allow us to detect resin depletion early. We evaluated other learning algorithms (kNN, SVM, and neural network) in order to detect resin depletion early while rejecting noise.

Reduced Salt Usage Figure 4.12 shows the average salt used by the softener system in Chadbourne over a three-year timespan starting in 2012. The dashed line represents the salt consumption of the building's water softener in pounds per day before installing AWESOME. The solid line shows salt consumption for the following year after AWESOME was installed. Following the installation of AWESOME in July 2013, the building's per-day salt consumption fell because the number of gallons between regenerations increased by more than 30%. Salt consumptions in the other buildings in which we piloted AWESOME also dropped. Figure 4.13 shows a comparison of the salt consumed by the water softeners in our three pilot buildings on the UW campus

before and after installing AWESOME. Chadbourne Hall showed the most improvements because that building's water softener system was configured to regenerate *nearly twice as frequently* as it should have, by comparison to factory parameters. The other buildings, though configured correctly according to factory parameters, were still using too much salt because their stock controller settings were far too conservative.

We think that the salt consumption in Chadbourne hall improved more relative to the other two pilot buildings because the hardness of the municipal water supply to that building is the lowest.

Other Benefits of AWESOME With the reduced volume of salt used to flush the water softener system in Chadbourne, there was a proportionate decrease in Chloride and Sodium pollution.

AWESOME makes it possible for maintenance staff to monitor the status of the water softener system from any computer. Water softeners are often installed in less accessed parts of the building as matter of aesthetic principle, however, this often makes access inconvenient and unwieldy. For example, before installing AWESOME , the only way to be sure the system was working properly was to climb a 6-foot ladder in a mechanical room in the basement of Chadbourne. Even then, it was not possible to get continuous historical data about the water flow through the softener.

SPOCK

In this section, we evaluate our forecasting methods and compare them to existing techniques for choosing when to regenerate water softeners. We will also compare our methods to an oracle forecaster, which knows exactly what the flow and hardness will be in the future.

Dataset

The dataset analyzed in this section consists of flow and hardness data collected from a single building by an AWESOME sensor during the months of January-June 2016. The building is a research lab on the University of Wisconsin campus

that employs about 100 people. The main water consuming activities in the lab are plant cultivation and cooling.

Sensor Value Forecasting

Here we evaluate our sensor value forecasting methods to characterize their accuracy.

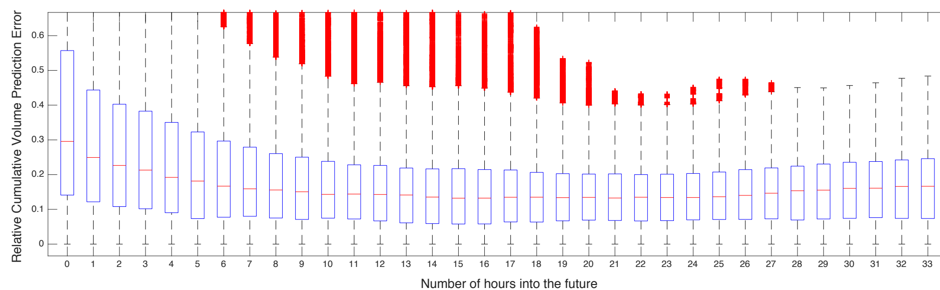

Figure 4.14: Relative prediction error of cumulative water flow (y-axis) as a function of the forecast horizon (x-axis). The predictions evaluated here are based on 14 days of historical data.

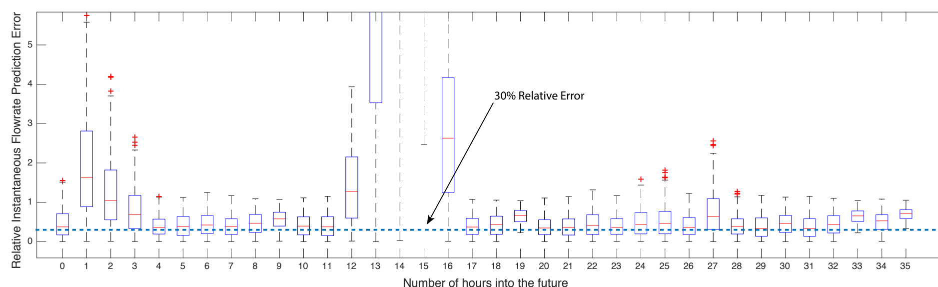

Figure 4.15: Relative prediction error of instantaneous water flow (y-axis) as a function of the forecast horizon (x-axis). The predictions evaluated here are based on 14 days of historical data.

Flow Forecasting We first evaluate our flow forecasting algorithm by analyzing its relative prediction error as a function of the number of hours ahead we are predicting. We define relative prediction error as

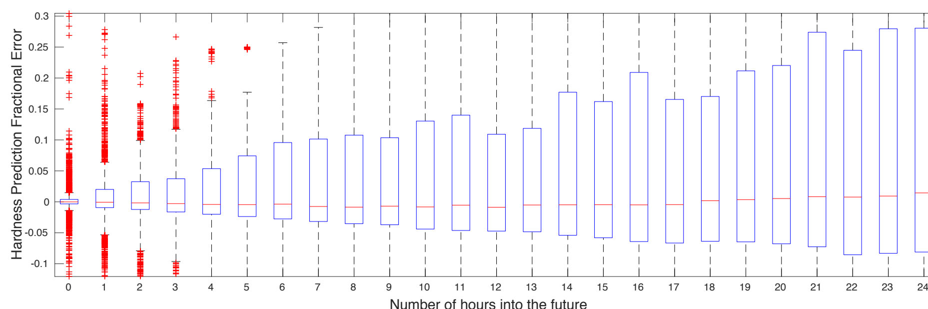

Figure 4.16: Relative prediction error of water hardness (y-axis) as a function of the forecast horizon (x-axis). The predictions evaluated here are based on 14 days of historical data.

$$e_{p_r}(h) = \left| \frac{p(t-h) - s(t-h)}{s(t-h)} \right|$$

where h is the number of hours in advance our algorithm is predicting, $p(t)$ is our prediction as a function of time, and $s(t)$ is the actual measured sensor value as a function of time. In this evaluation, we use roughly 14 days of historical data to make forecasts of 1-1.5 days in advance. Our computations of relative prediction error for cumulative flow are shown in Figure 4.14. Computations of relative prediction error for instantaneous flow are shown in Figure 4.15.

Relative prediction error for cumulative flow is on average below 30% for all time horizons we studied, and it actually decreases for longer time horizons. The instantaneous flow signal tends to be very bursty—as fixtures turn on and turn off in the building, water flow starts and stops sporadically. If these bursts do not arrive at the exact moments that our algorithm expects them to, but instead arrive several minutes before or after, then the near-term cumulative flow error will be higher. As long as the bursts of flow eventually arrive, the long-term cumulative flow is low because it captures all the flow that has happened over a long period of time, and the errors in the prediction cancel each other out.

The instantaneous flow error, however, does not benefit from this error canceling effect. If flow bursts do not arrive exactly as expected, the instan-

taneous flow error will be high. This does not actually hurt us though, and our predictions are good enough. As we will see later, it is not the absolute magnitude of forecast error that matters, but the timing of peaks and valleys in the cost function. As long as our algorithm can accurately predict the timing of peaks and valleys in instantaneous water flow forecasts, the later steps will work properly.

Hardness Forecasting Our hardness forecasting algorithm is intended to predict whether or not water samples will read hard in the next 24-hour time horizon. Since our water hardness sensor readings are sparse—long stretches of zero readings followed by short periods of hard water readings—we are interested in predicting not only the exact value of the hardness sensor readings but also the presence or absence of hard water. Our goal is to correctly predict the value of the sensor reading, if possible. If, during a forecasting horizon, we do not correctly predict the hardness value, but we do correctly identify the presence or absence of a hard water event, then the algorithm has largely achieved its goal.

A plot of the relative errors of SPOCK's hardness prediction algorithms is shown in Figure 4.16. On average, the relative prediction errors are close to zero for time horizons up to 24-hours in the future. The variance, however, increases for predictions made further into the future.

A receiver operating curve for our hardness forecasting algorithm is shown in Figure 4.17. We generated the ROC by varying the threshold at which we identified the water as being hard—higher thresholds resulted in a higher false positive rate. Our dataset did not include sufficiently high hardness values to generate true positive rates above 0.9 because AWESOME, the system used to gather the data, regenerated the water softeners before the hardness reached high values.

Comparison to Reserve Capacity and Oracle Algorithms

Here, we compare SPOCK to the oracle forecaster and the reserve capacity algorithms. The oracle forecaster algorithm is the same as SPOCK, except that

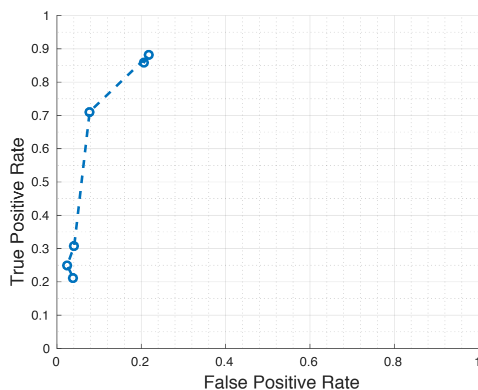

Figure 4.17: Receiver operating characteristic for SPOCK's hardness forecaster.

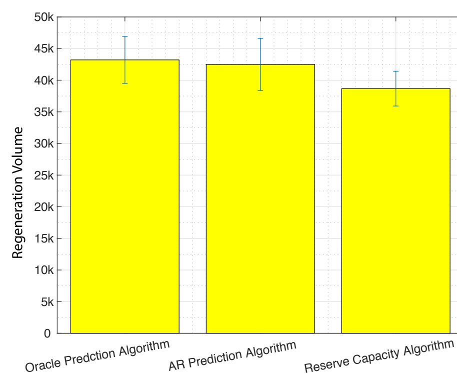

Figure 4.18: Comparison of tank capacity using (i) oracle scheduler, (ii) SPOCK autoregressive scheduler, and (iii) reserve capacity. SPOCK performs nearly as well as the oracle predictor, which knows exactly what the sensor readings will be for all future readings.

the forecasting algorithm uses real data in the dataset rather than trying to predict future sensor readings. The reserve capacity algorithm is the method popularly used to control water softeners, described in Section 4.1.

Figure 4.18 shows a bar chart comparing the average number of gallons between regenerations for each of the three algorithms (higher numbers are better). Table 4.4 gives percentage improvements over the reserve capacity algorithm. SPOCK's autoregressive prediction algorithm outperforms the reserve capacity algorithm by roughly 10%. Furthermore, SPOCK performs only about 2% worse than the oracle prediction algorithm, which has access to much higher quality predictions about future sensor readings. Despite having access to imperfect sensor predictions, SPOCK can still make reasonable control decisions. Since our timescale of interest for scheduling regenerations is on the order of hours, even some errors in flowrate and hardness predictions can have negligible effects on the timing of control events. We acknowledge that this may not be the case for other building automation tasks, such as controlling

Algorithm	Improvement over Reserve Capacity
Oracle	11.7%
SPOCK	9.9 %

Table 4.4: Improvements of the oracle algorithm and SPOCK over the reserve capacity.

light levels in rooms. How the timing of critical building automation events is affected by the accuracy of sensor predictions is still an open question.

4.5 SUMMARY OF AWESOME AND SPOCK

Awesome and SPOCK are complementary systems we developed to monitor and control the behavior of a building's water treatment systems. These systems, which are normally not equipped with closed-loop controllers, are often wasteful users of water and other treatment chemicals. Awesome monitors water quality in real time to determine when water treatment systems need to be regenerated. SPOCK forecasts water consumption patterns to schedule future regenerations.

Emonix, Hot/Cold, Awesome and SPOCK constitute a comprehensive testbed for building management systems. Each is built with open-source hardware and software, making it easy to test new functionality or modifications.

5 HERMES: A HYPERVISOR FOR MOBILE AND IOT DEVICES

Modern embedded sensing and mobile applications increasingly perform diverse functions, including displaying user interfaces, managing networking, performing real-time data acquisition, and more. Some even allow third-party code to be downloaded and run alongside the factory firmware [105]. Such diversity in runtime requirements poses challenges to software architects, who must manage the often competing needs of different tasks.

To manage the diverse runtime requirements of embedded software, we have developed a lightweight embedded hypervisor we call Hermes [1] , targeted to ARM Cortex-M microcontrollers. Other authors have since proposed similar systems for mobile phones [32, 39, 86] and microcontrollers [90], but Hermes remains the first and only complete virtualization environment for MMU-less processors at the time of this writing.

IoT applications are frequently implemented on CPUs without an MMU in order to save cost and power. While the cost of MMU-equipped Linux-capable processors is going down all the time, energy considerations (especially for mobile applications) are not likely to go away.

The problem we set out to solve is one of I/O latency in such a complex runtime environment. Real-time OS scheduling algorithms cannot guarantee deadlines will be met under high I/O load. People usually solve this problem by running time-critical operations on a separate CPU [84]. For example, high-frequency signal sampling may be implemented in bare metal code running on an independent microcontroller while the user interface, networking, storage, etc. runs on the main device. This approach has a lot of obvious shortcomings: increased hardware and software complexity, power consumption, physical size, verification difficulty, etc.

In this context, "real-time" refers to the schedulability of *user-level code*—the OS has no ability to schedule interrupt service routines triggered by asynchronous I/O events [81], which are managed by the CPU hardware and

[1] **H**yp**E**rvisor for **R**eal time **M**icrocontroll**E**r**S**
`http://hermes.wings.cs.wisc.edu`

interrupt controller. The RTOS can re-order the processing of interrupt service routines, but it cannot prioritize user-mode code above ISRs; as we will see, it is this lack of flexibility that causes non-deterministic I/O latency.

Driver-level I/O processing has traditionally been assumed to be a negligible component of overall response time—an assumption that was valid 30 years ago as these real-time scheduling algorithms were being developed. At that time, embedded computers were single-purpose machines that largely performed the same task repetitively.

But that assumption of single-purposeness is becoming less valid. Modern microcontrollers are equipped with a rich set of peripherals that was unimaginable in the 1980s.Network interfaces, high-speed data acquisition devices, touchscreens, and more all have a diverse range of requirements, but they are treated the same by the RTOS and CPU. Exception management for low-priority I/O is always performed before user-mode code can respond to high-priority events, creating a kind of unintended priority inversion (depicted in Figure 5.1). Consequently, response times to latency-sensitive I/O events are not deterministic, which can result in failure. We explored this problem in [68].

Conventional wisdom among real-time programmers is that ISRs should be as short as possible: clear the interrupt, maybe transfer a few bytes of data, and exit. Userland code should be responsible for responding to the event. In a crowded software environment with multiple drivers and tasks competing for CPU time, this programming method has the effect of delaying the actual response of all I/O events until all ISRs have finished executing. These delays break the assumptions that underlie real-time scheduling algorithms, which require the highest-priority task to always run first. Instead, we are running the driver code associated with low-priority tasks before the user code for high-priority tasks, and RTOSes do not have flexibility to change this behavior.

Hermes is a lightweight virtualization platform that lives between the hardware and the operating system. At its core, Hermes consists of some initialization code and a single interrupt service routine that catches and preprocesses all exceptions before dispatching them to the operating system. In its role as a mediator of exception processing, Hermes can allow multiple operating systems or bare metal applications to run side-by-side on an MMUless

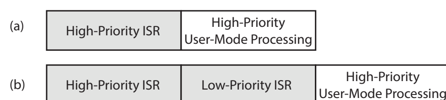

Figure 5.1: Timeline of (a) high-priority ISR followed by user-mode I/O processing and (b) low-priority ISR co-occurring with a high-priority ISR, delaying high-priority user mode processing.

microcontroller, dispatching exception processing to the appropriate OS as necessary, much like a hypervisor running on a PC or server. There are several advantages to this software architecture:

1. **Performance.** For time-critical applications, Hermes can provide a thin layer between the software and the hardware. Real time operating systems (RTOSes) on the other hand, often come with a lot of overhead in the form of system call latency for time-critical tasks. This may be unacceptable in applications where time-critical tasks need to coexist with other less critical code like networking or user interface software.

2. **Privacy.** Because it lives between the hardware and the app, Hermes is in a unique position to intercept and anonymize personal data, coded in I/O transactions. We are exploring this topic in another line of work [71].

3. **Portability.** Hermes can provide a consistent virtual environment for all higher level software, regardless of the underlying hardware. This could enable, for example, edge computing devices with heterogeneous hardware implementations to run user apps targeted to a common platform.

A diagram of the Hermes software architecture is shown in Figure 5.3. We are implementing Hermes on an ARM Cortex M7 CPU called the Atmel SAM E70 [34, 35] which has 2 Mbytes of flash and 384 kbytes of RAM. The ARM Cortex M7 core uses an instruction set called Thumb-2, which is a simplified version of the ARM instruction set, allowing most instructions to be encoded in 16 bits. Other Cortex-M cores (M3, M4, etc.) also use the Thumb-2 instruction set and have a similar programmer-visible architecture, which should make it

possible to port Hermes to these other models. It also includes many advanced features of the latest ARM microcontrollers such as a floating point unit, a memory protection unit, separate instruction and data caches, and many peripherals. We have tested Hermes by running a FreeRTOS v9.0.0 [23] guest on top of the hypervisor. The contributions made by this work are the following:

- We describe in detail the implementation of a hypervisor built for real time software environments to improve responsiveness of real-time tasks.
- We discuss challenges of implementing a hypervisor on a hardware platform with minimal support for virtualization.
- We evaluate the performance of our hypervisor running multiple real-time tasks in a realistic mobile environment, comparing it to FreeRTOS.

5.1 BACKGROUND

Real-time operating systems have been around for a long time, and we have had a lot of opportunity to study their schedulers. One may wonder, *can we achieve the same results by modifying an RTOS?*

In principle, we could, but doing so makes the system much more difficult to program. Figure 5.2 outlines a sequence of events needed to implement our techniques in an RTOS. The key modification is that the ISR would need to disable the scheduler after disabling interrupts (between the first and second bubbles). This would prevent the RTOS scheduler from switching to a new task, which is the cause of the deadlock. The modification would need to be made to the tasks and the drivers, not the internal RTOS code. These modifications would be application specific—we would only make these modifications in the highest priority drivers and associated user code. There may need be some modifications to the way the interrupt controller is configured, but they would be relatively minor. We have not experimented with this.

The reason that we consider this a bad solution is that it requires programmers to observe extra rules to prevent deadlocks in their apps. Real time app programming already has specialized rules (eg. no API calls in high-priority

ISRs, etc.) that are generally unknown to most non-real time programmers. This technique would increase the difficulty of writing real-time apps, increase the prevalence of bugs, and require programmers to have deeply specialized knowledge of issues around real-time systems programming. Most regular mobile programmers probably do not even know what real time programming is, so it would be unrealistic to expect them to have deep knowledge of the specialized techniques that it requires.

Furthermore, the RTOS solution would make the performance of each individual app mutually dependent on the set of other apps with which it shares the CPU. As new apps are loaded to a device, the performance of real-time apps might be degraded.

By contrast, the hypervisor solution creates independent virtual runtime environments for each VM which are not affected by other VMs that share the CPU. Each VM in Hermes consists of a group of ISRs and user code. Hermes independently emulates the CPU's interrupt controller, system timer, and other hardware blocks for each VM. By contrast, RTOSes only keep track of the program counter and stack pointer for each running task. Because Hermes has independent control of the state of the interrupt controller for each VM, it can handle enabling and disabling interrupts as the CPU transitions from ISRs to user code, a job that would otherwise be delegated to the individual tasks. This allows us to write and test the code once in Hermes. App programmers do not need to worry about enabling and disabling the scheduler, which makes programming much more intuitive and reduces the possibility of introducing bugs.

We considered several simpler alternative solutions that could be implemented in the RTOS to alleviate I/O latency:

Use the interrupt controller to statically mask low-priority interrupts while executing real-time tasks. Figure 5.2 outlines the sequence of events needed to mask low-priority interrupts during a high-priority event. Before any exception has been raised, we want all interrupts to be enabled. We would only want to mask low-priority exceptions *after* a high-priority exception has been raised. Low-priority exceptions would then remain masked while the

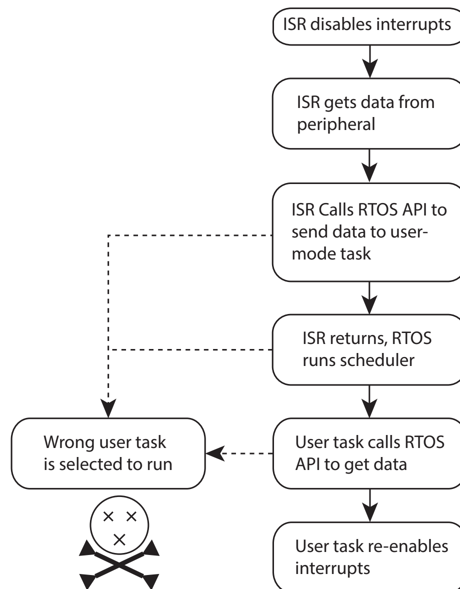

Figure 5.2: Sequence of events required in an RTOS to disable low-priority interrupts in a high-priority ISR. Interrupts would ostensibly be re-enabled in user mode after the I/O event has finished processing. However, the RTOS could transfer control to a different task, which could cause a deadlock.

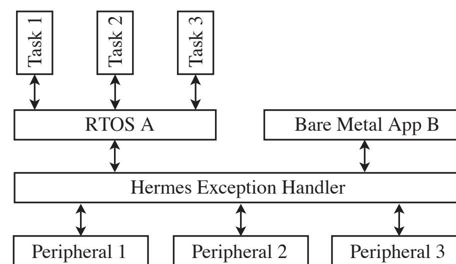

Figure 5.3: Architecture of the Hermes Hypervisor. The main component of Hermes, its monolithic exception handler, intercepts all exceptions before dispatching them to the guests.

high-priority ISR is executing and until the associated user-mode code has finished.

This requires disabling interrupts in an ISR, processing the I/O event, and re-enabling interrupts in the user-mode task associated with that event. In this scenario, the critical section of code—the code during which interrupts are disabled—spans a return from exception. For this to work reliably, the ISR must return immediately to the associated user-mode task that collects and processes the data from the ISR. This is dangerous because after the return from

exception, before executing user mode code, the RTOS will run the scheduler and possibly select a different user-mode task to run.

In fact, there are three points in this sequence of events where the RTOS's scheduler could select a different user-mode task to run[2]. If that happens, the different task may not be aware that the interrupt mask level that was set by the ISR, which may result in a deadlock. Hermes solves this problem by creating a truly isolated execution environment for every guest. Hermes tracks and emulates the CPU state for each guest, including the interrupt mask, interrupt controller, and many other peripherals. When a context switch occurs, Hermes restores the new guest's full emulated CPU state, not just the guest's program counter and stack pointer, as is done in an RTOS. This avoids the problem in Figure 5.2 where an incorrect CPU state set by an ISR can follow the execution stream into the wrong user-mode task.

Because we are emulating the CPU state independently for every guest on the system, there are fewer rules that programmers need to follow, and there is less possibility of adverse interaction among tasks.

Disable interrupts in user-level code. This would allow us to process the I/O event in userspace without interruption. It does not solve the problem of ISR-userspace latency, since more than one exception may execute sequentially before user code gets a chance to disable interrupts.

Process I/O events in the ISR. This would allow us to ensure that our I/O events are processed in a timely fashion. This could be an acceptable solution for a single-purpose bare-metal app with no other tasks running concurrently. The problem with this approach in an RTOS is that it monopolizes the CPU during the entire I/O operation, likely causing other tasks—even higher priority ones—to hang while the I/O event is handled.

Re-prioritize the interrupts. We could use the CPU's interrupt prioritization circuits to execute the time-critical ISR first, before other ISRs. This wouldn't

[2]The FreeRTOS documentation instructs users not to call RTOS APIs inside critical sections to avoid deadlocks for this reason.

decrease latency in an RTOS environment because lower priority ISRs will always execute before the user space code.

Disable low-priority interrupts inside high-priority ISRs. When we process a high-priority interrupt, we could disable all lower priority interrupts. This would cause the high-priority ISR to return directly to user-mode code without processing low-priority ISRs. This seems like a hack because it requires the driver code to be tightly coupled to the application: if threads are added, removed, or re-prioritized, the driver code would also need to be modified to reflect the changes in prioritization, likely resulting in bugs. The hypervisor model allows us to think about tasks independently without worrying about complex interactions between user-mode code and drivers.

None of these solutions is a viable alternative because they cannot reduce latency while maintaining a responsive runtime environment for other concurrent tasks.

5.2 ARCHITECTURE

Generally speaking, there are two classes of problems a hypervisor needs to solve in order to create a virtual runtime environment for its guests. First, it needs to emulate I/O operations in software such that guests think they have access to I/O ports and peripherals even though they do not. Second, it needs to emulate privileged instructions and memory accesses such that guests think they can execute privileged instructions even though they can't.

In both cases, guests should not be able to directly change the physical state of the CPU, for example by masking interrupts or putting the device in a low-power sleep state, because these operations could affect the execution of other guests. Instead, the hypervisor must emulate these privileged operations in software so they affect only one guest, not the state of the entire system.

Interrupt Hooking Used to implement keystroke loggers in early PCs, interrupt hooking is a technique by which we modify an interrupt vector table entry for a particular I/O event, redirecting it from the operating system's

interrupt handler to a different function called an interrupt hook. When the I/O event occurs, the interrupt hook function runs first, inspecting, modifying, or logging data generated by the I/O event. When it finishes, the interrupt hook then calls the operating system's interrupt handler which does not know that the interrupt hook ran ahead of it. This is the basic concept used by Hermes and other hypervisors to emulate I/O events.

Hermes is a single monolithic interrupt service routine that intercepts all CPU exceptions before they can be processed by the operating system. Figure 5.3 shows a diagram of the interactions between the Hermes hypervisor and its guests. On boot, the Hermes initialization code sets up the CPU's exception table to point to the Hermes ISR. It then launches the guest operating systems in the ARM CPU's unprivileged execution mode[3]. Broadly, there are three kinds of exceptions that Hermes needs to handle:

Faults are exceptions caused by software running on the CPU. In general, faults are caused by privileged instructions or memory operations being run in unprivileged mode. When a guest causes a fault, it is a cue to Hermes that the guest's virtual execution state needs to be modified. For example, when a guest attempts to return from an exception, it causes a fault because the instruction it uses is privileged, and the guest is running in unprivileged mode. The Hermes exception handler will trap the fault and modify the guest's virtual state to indicate that the guest is running in virtual unprivileged mode. The different kinds of faults that Hermes handles are explained in detail below.

Direct I/O Interrupts are interrupts triggered by peripherals that are exclusively owned by one virtual machine. An example of this would be a hardware serial port. Hermes handles all direct I/O interrupts in the same way: by passing control to a hardware-specific ISR in the appropriate guest. There is no specialized code for individual direct I/O interrupts. Hermes handles these exceptions by setting up a virtualized exception stack frame for the guest which emulates the stack frame that would have been created by the hardware if the guest VM were running directly on the bare metal. It then returns the CPU state to unprivileged thread mode and passes control to the guest's interrupt service

[3]Normally, operating system code would run in privileged execution mode, but when the RTOS is running as a guest inside Hermes, it executes in unprivileged mode.

routine. The guest's ISR will then process the exception. As it does so, it will perform privileged operations such as accesses to privileged memory regions, execution of privileged instructions, and exception return. These privileged operations will be trapped by faults and emulated by Hermes.

Indirect I/O Interrupts are interrupts triggered by peripherals that may be shared among multiple guests. An example of this would be a bridged Ethernet interface where the NIC hardware can be shared by multiple guests with different virtual MAC addresses. These are different from direct I/O interrupts because the driver code for different peripherals must be inside the Hermes hypervisor. Indirect I/O interrupts are associated with peripheral devices that are emulated by the hypervisor, so the hypervisor must also store some virtual state information for each guest VM that uses the emulated peripheral. In general, Hermes will expose some virtualized hardware interface to the guest, which is either a full or simplified version of the underlying hardware peripheral's interface. When the guest modifies the virtual peripheral's state, Hermes will record those changes in its internal data structures and use the guest's settings in future interactions with the hardware peripheral.

A key difference between direct and indirect I/O interrupts is where the peripheral-specific driver code lives. For direct I/O exceptions, the driver code is part of the guest, and indirect I/O exceptions, the driver code is in the hypervisor. A consequence of this is that for shared peripherals using indirect I/O exceptions, the Hermes driver may have to significantly modify the hardware peripheral's settings to make the virtualized peripherals behave as expected from the guest's perspective.

Sleep: If a guest has nothing to do and needs to wait for an I/O event before it continues processing, it can put its virtual CPU into a sleep state using the standard ARM sleep instruction `wfe`. This is the same method an app would use to put the CPU into a low-power sleep state if it were running on the bare metal. When a guest goes to sleep, the scheduler will be notified, and it will not be scheduled to run until it gets an I/O event from a peripheral that it owns. Other guests will then be allowed to use its share of the CPU. There are several types of faults that Hermes handles:

Privileged register access generates a (possibly imprecise) bus fault. These are normally associated with attempts to access peripherals. The Hermes bus fault handler either emulates the access in software (if supported) or executes the instruction directly. At the moment, we have only implemented emulation of a small subset of the privileged registers, most of which are related to critical functions like setting the vector table offset register or exception prioritization.

Privileged instruction generates a usage fault. These are associated with attempts to change the execution state of the processor (modifying stack pointers, exception masking, etc.). Hermes never directly executes privileged instructions. Instead, it decodes the instruction and modifies the guest's emulated state by updating fields in a data structure.

SVCall handler is a stub that just jumps to the guest's SVCall handler, which is executed in unprivileged mode.

I/O Exceptions handler is a stub that jumps to the guest's implementation of the handler for that exception, which is executed in unprivileged mode. Hermes looks up the address of the guest's handler in the guest's exception table. If the guest attempts to execute a privileged instruction or access a privileged memory region from within its exception handler, that action will trap to the Hermes exception handler so it can be emulated.

Guest return from exception creates a memory management fault. Hermes catches the fault and returns execution to the guest's user mode code.

Establishing Guest Priorities

The ARM Cortex-M interrupt controller includes an interrupt priority mask register called `BASEPRI` that can disable interrupt sources lower than a given priority. The mask register is normally set by an RTOS to enter/exit critical code sections. Hermes does not allow guests to directly set the architectural `BASEPRI` register—that is a privileged operation that Hermes emulates in software. Instead, it maintains a virtual `BASEPRI` register for each guest which is used as the guest's priority.

If a guest elevates its `BASEPRI` register to disable certain interrupt sources, Hermes elevates that guest's priority among the pool of currently active guests.

At any given time, the guest with the highest `BASEPRI` value will be given access to the CPU. Under some circumstances, Hermes may also set the architectural `BASEPRI` register to the value configured by the guest, disabling interrupt sources for low-priority guests. This is how we establish virtual isolation among guests—by disabling low-priority interrupt sources and maintaining separate states for the virtual interrupt controller in each guest. In contrast, operating systems—real time or otherwise—do not maintain separate virtual hardware states for different tasks or processes. Instead, OSes prefer to provide programmers with APIs and drivers to interact with the physical hardware. Nothing in principle prevents them from virtualizing memory or I/O in the way that hypervisors do. It's just that our custom is to c all runtime environments that manage separate virtual state for hardware peripherals hypervisors and runtime environments that do not manage virtual state for peripherals operating systems. So the question about whether we can modify the operating system to get the real-time behavior that we want is really about the architectural and philosophical differences in implementation between these two runtime environments. Probably if we redesigned operating systems today from the ground up using modern hardware and software, we would end up with a design somewhere between an operating system and a hypervisor: more virtual state than an OS, but more handholding for I/O (like drivers) than a hypervisor.

Opportunities

Running a hypervisor on embedded IoT equipment enables some interesting possibilities for IoT software.

Distributed Processing on a Single Chip Many embedded hardware designs use a distributed computation model to separate a complex task into several independent execution environments. For example, a board might have one network processor, one sampling processor, and a main CPU, each performing its own specific task independently of the others. This type of design complicates the hardware and software and likely drives up the cost, size, and energy requirements of the equipment. With a hypervisor, we can

run all software on a single CPU while maintaining isolation by running each independent application in its own VM. CPU and resource allocation can be strictly controlled by the hypervisor to ensure that deadlines are met.

Security and Privacy Mobile device privacy is an interesting area of research in which we try to limit the amount of personally identifying information that users divulge about themselves in the course of using mobile and IoT apps. In a parallel line of investigation, we have used Hermes as a platform for capturing information about users in mobile apps that is collected from sensors like video cameras and microphones on board mobile and IoT devices [71]. The sensor data streams captured by Hermes can be processed by anonymization software in real time before being handed off to the appropriate app. The advantage of operating system and hypervisor-level techniques over the more common network middlebox approach [36, 40] is that the sensor data is unencrypted at the hardware level. A hypervisor-based privacy agent does not need to worry about decrypting and SSL stream to inspect or modify its content.

Authenticating the software on an unattended embedded device is still an open problem. A few proposed solutions [27, 88] rely on measuring the timing of some arbitrary computational operation. The hypervisor may be able to serve as a root of trust for virtualized applications by implementing a virtualized trusted platform module (vTPM) [22] to be used by underlying software components. It may be possible to implement a virtual TPM in software using either ARM TrustZone [95] or an on-chip cryptographic accelerator [46].

Loadable Apps Apps written by users or third parties could be easily selected from an app store and run natively on the IoT device, allowing flexibility to the end user without overloading the cloud services. A major challenge for this model of app distribution is that systems without an MMU must have all their code compiled together before runtime. Third party apps, when loaded dynamically, could overwrite each other's memory regions during execution. Furthermore, allowing third party apps to run on an IoT platform creates software versioning headaches because we must make sure that the OS and

user app share the same library and system call implementation: when we upgrade one, we must upgrade all.

A hypervisor could solve both problems by running each user app inside an independent virtual machine. When we need to context switch away from one user app, we can snapshot it into a peripheral memory device (external SRAM or flash) and load a different app into the same memory space. This would allow us to run multiple apps with a single virtual memory space without the use of an MMU. From the app's perspective, it would be running on an unshared virtual machine, and it would not need to share common drivers with the RTOS, making software compatibility much simpler.

Challenges

By running a hypervisor on a Cortex-M CPU, we are using the device in a way that was not intended by its designers. Emulation of privileged instructions and memory regions, I/O, etc. exposes some interesting features, optimizations, and design decisions in the CPU that programmers would not normally encounter.

Compile-Time Guest Setup Since we are dealing with a system that has no MMU, we are required to compile all guests with the hypervisor into a single runtime binary. The practical challenge is that, for symmetric guests (more than one instance of a single guest OS), we must change the name of each function and variable in order to avoid linker errors. This can be mildly annoying because it makes the RTOS code harder to read. We have written a script to perform this task automatically.

Imprecise Bus Fault Exceptions The ARM Cortex M line of CPUs throws bus fault exceptions for accesses to privileged memory regions that are mapped to certain control registers. Some of these exceptions can be imprecise, meaning that the CPU does not record the exact instruction that caused the exception. Instead, it will throw a bus fault as soon as possible (in our experience 2-10 instructions past the faulting instruction). This makes the job of the Hermes exception handler difficult since it does not know which privileged memory

access needs to be emulated. The only thing we can be sure of is that the faulting instruction occurs earlier in the instruction stream than where the exception was thrown.

We solve the problem of imprecise bus fault exceptions by tracing back through the instruction stream to look for a privileged instruction with the correct effective address that is likely to have caused the imprecise exception. Starting at the address of the instruction that caused the exception, we trace back through the last five instructions in order of memory address. We decode each instruction and compare its effective address to the address that caused the bus fault. If the instruction's effective address matches the offending effective address, then we assume a match and emulate that instruction.

If we do not find a bus fault address match in the last five instructions before the exception was thrown, we examine the same five instructions again, this time looking for any instruction that could have caused a bus fault. We do so by examining the effective address of each instruction and determining if it corresponds to a privileged memory region. We assume the most recently executed instruction that accessed a privileged memory region was the one that caused the bus fault.

Clearly, there are some pathological cases that could cause this approach to fail. For instance, consider the following instruction sequence for updating the value of a privileged hardware I/O register, executed in unprivileged mode:

```
  ld   r1,=privilegedAddress
* ld   r0, [r1]
  add  r0, r0, #4
* st   r0, [r1]
```

In this instruction sequence, the `ld` and `st` instructions marked with a * will each generate a bus fault because they access a privileged memory region. The bus fault may be imprecise, and it may be delayed by up to ten instructions past the offending instruction. This means that the CPU may not take an exception for the `ld` instruction until several instructions past the `st`. Since both instructions have the same effective address, we won't be able to trace back through the instruction stream and distinguish the two instructions based on

Instruction	Function
`mrs`, `msr`	Read and write to special-purpose registers that control the execution state of the CPU. These instructions are used by the OS to set the supervisor and user stacks, change between supervisor and user mode, etc.
`wfe`, `wfi`	Put CPU to sleep and wait for interrupt.
`cpsie`, `cpsid`	Enable/disable interrupts.

Table 5.1: List of privileged instructions that do not cause privilege violations when executed in unprivileged mode.

effective address only. Furthermore, by the time the exception is taken, the value in register `r1` may have been changed by subsequent instructions, making it impossible to identify the effective address of the offending instructions.

Unfortunately, this instruction pattern is fairly common in code that configures or modifies the settings of peripherals or I/O devices. Our solution to this problem is to add pipeline synchronization instructions after each privileged memory access, which forces all in-flight instructions to complete before proceeding. This forces any exceptions caused by in-flight instructions to be taken before proceeding, making it possible for Hermes to identify the exact instruction responsible for the privileged memory access. Hand-patching OS kernels is a manual and painful process. A pre-patched version of FreeRTOS is available on the Hermes website. We have not yet developed any automated tools to search for privileged memory accesses in the OS source code and automatically add pipeline synchronization instructions in the right places, although it would in principle be possible to do so.

So far, we have not encountered any code in a guest that causes this approach to fail to emulate the guest.

Some Privileged Instructions Don't Cause Exceptions When Executed in Unprivileged Mode Some privileged instructions on the ARM Cortex M7 do not cause privilege violation exceptions when executed by the guest. A list of some privileged instructions that we know do not cause exceptions when

executed in unprivileged mode is shown in Table 5.1. This is a challenge of not having hardware support for a hypervisor. For example, the `mrs` and `msr` instructions are classified as privileged instructions, but when they are executed by code running in unprivileged mode, they fail silently: the register write is not committed, and the processor continues normal execution. Oddly, whether or not we are allowed to execute `mrs` and `msr` instructions seems to depend not on the privilege mode of execution, but on the stack pointer we are using. If we are using the Master Stack Pointer (MSP), we can execute `mrs` and `msr`, but if we are using the Process Stack Pointer (PSP) we cannot. Usually (but not always) the MSP is used when executing privileged code and the PSP is used when executing unprivileged code.

The problem is that if a guest OS tries to modify the processor state with one of these privileged instructions, that state modification cannot be registered by Hermes since it does not cause an exception. The privileged instruction will complete like a `nop` instruction without modifying the CPU state. Critical CPU state changes like disabling interrupts will not work as intended.

FreeRTOS FreeRTOS is a popular (if not the most popular) real-time operating system for embedded and IoT computers. It has been ported to CPUs manufactured by 20+ manufacturers representing every commonly used architecture (ARM, x86, etc.). FreeRTOS implements a rate monotonic scheduler in which each task has a fixed priority, and the highest priority task that is ready to run is executed first.

FreeRTOS and others like it impose strict rules about how user mode software can call the RTOS API in order to avoid deadlocks such as in (1). For example, no RTOS API calls can be made within high-priority ISRs, presumably to maintain responsiveness for the rest of the system. This further limits the scope of what we can do with the RTOS.

ARM Exception Handling The ARM Cortex-M CPUs include an interrupt controller that allows software to prioritize interrupts. Each interrupt can be assigned a priority between 0 and 255 (higher priority values are associated with

higher priority). ISRs associated with low-priority interrupts can be preempted by ISRs for high-priority interrupts.

Software running on the ARM device can also mask exceptions below a desired priority value by setting a core register called `BASEPRI`. When an interrupt occurs, its priority is compared with the value in `BASEPRI`, and its ISR is only executed if the interrupt priority is greater than the `BASEPRI`. Otherwise, the ISR is delayed until the value in `BASEPRI` is reduced below the interrupt's assigned priority. Changing the value in `BASEPRI` is done by executing a privileged `msr` instruction.

We circumvent this problem by patching the OS kernel, adding an undefined instruction immediately following an `mrs` or `msr`. When the hypervisor encounters an undefined instruction exception, it will search backward in the instruction stream for an `mrs` or `msr` instruction and emulate it. If we run an unpatched kernel inside the hypervisor, it will crash because the intended CPU state modifications will not happen as intended.

In FreeRTOS, there is a total of 38 assembly language instructions that need to be added in order to make the OS run as a guest in the hypervisor. All of the additional instructions are in FreeRTOS's chip-specific code that deals with timer interrupts and scheduling. We have posted a patched version of the FreeRTOS v9.0.0 kernel on the project website.

Online Instruction Decode Because some privileged memory accesses cause imprecise exceptions, it was necessary for us to identify the likely source of an imprecise bus fault by sequentially decoding instructions immediately prior to the bus error exception and searching for one with a matching effective address. Our instruction decoding code, works to correctly execute a guest running FreeRTOS v9.0.0.

However, this approach is slow, and there is probably a better solution.

We Invalidate I$ for Every Privileged Instruction The way Hermes handles un-emulated privileged instructions is by copying the instruction from the guest's instruction stream into a subroutine that is called by Hermes. Before

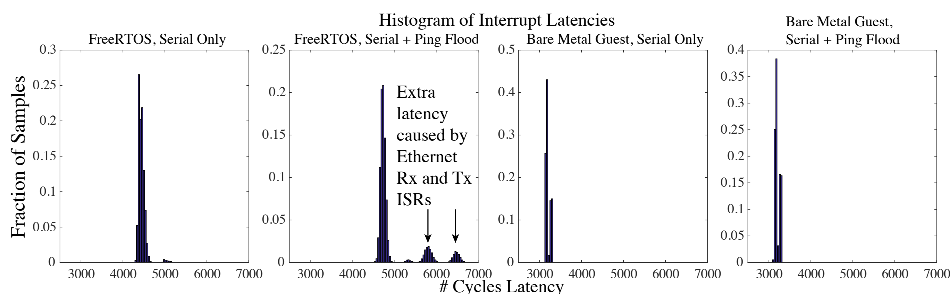

Figure 5.4: ISR-userspace latency histograms. Latency is measured as the number of cycles elapsed between executing the serial port receive ISR and beginning of userspace processing.

we call the subroutine, we must invalidate the CPU's instruction cache because it contains stale data from before the instruction was copied to it.

The ARM Cortex M7 core does not allow us to selectively invalidate lines in the instruction cache. In order to clear the stale data from the instruction cache, we need to invalidate the entire cache, which causes a huge slowdown in execution.

Privileged instruction execution is mostly associated with I/O events, and most privileged instructions appear at boot time when peripherals are being initialized. This could be a problem if we try to emulate an I/O device that requires a lot of reads and writes to privileged memory regions, as illustrated by the study of I/O performance in Table 5.4 in Section 5.3.

5.3 EVALUATION

Problem Validation

Using performance counters on the ARM Cortex M7 CPU, we measure the ISR-user space latency—the time between beginning of ISR execution to beginning of userspace data processing. This is a metric of how long it takes to respond to an I/O event. Ideally, for time sensitive I/O this time should be short and deterministic, meaning the same for each I/O event. We find that in the FreeRTOS environment, the ISR-user space latency is **less deterministic**

Software Environment	Entropy of Latency
FreeRTOS, Serial Only	2.73
FreeRTOS, Serial + Ping Flood	3.65
Bare Metal Guest, Serial Only	1.94
Bare Metal Guest, Serial + Ping Flood	2.08

Table 5.2: Entropy of the distributions of latency measurements (distributions shown in Figure 5.4). Low values of entropy are more deterministic. The bare metal guest running in Hermes has much more predictable latency than tasks in FreeRTOS. Under Hermes, latency is still highly deterministic under high I/O load.

under high I/O load, as expected. We have also experienced this problem when developing other systems, but we did not study it as carefully [74].

Experimental Setup

We measured the ISR-userspace latency for a serial port receive in FreeRTOS and Hermes. In FreeRTOS, we used an OS queue to transfer the data from an ISR to a user-mode task. In Hermes, we ran a FreeRTOS guest alongside a bare metal guest that transferred data between an ISR and userspace code using a memory buffer. In both runtime environments, we had two other periodic FreeRTOS tasks running alongside the latency test. In both cases, we ran the latency test in isolation as well as in the presence of high I/O load (a ping flood) to test how well each software environment could provide a deterministic runtime environment. The networking software that responded to the pings was implemented as a low-priority task in FreeRTOS for both environments.

Results

Figure 5.4 shows the results of our latency tests. Each subplot is a histogram of ISR-userspace latencies. Ideally, we would want these plots to have only one bar—a single response time for every I/O event. Figure 5.4 (a) and (b) show latency in FreeRTOS only, under low and high I/O load respectively. Under high I/O load, the latency histogram is more spread out because exceptions raised by unrelated I/O events delay execution of the user mode code in an

unpredictable way. This happens when a serial port exception and a network port exception occur close in time. Both exceptions must be processed before the user mode code to handle the serial port receive can begin executing. We get shorter and more deterministic response times when the serial port exception occurs in isolation. If the network port exception occurs near the same time as the serial port exception, the network port ISR will have to execute before the CPU can return to user mode, delaying the response time. This is an inherent disadvantage of running multiple unrelated programs on a single processor which we are trying to correct with Hermes.

Figure 5.4 (c) and (d) show the latency of the same I/O operation running as a bare-metal guest inside Hermes. Determinism is higher for histograms that are more clustered around a single value and lower for histograms that are more spread out.

Discussion

The reason that ISR-userspace latency is more deterministic in Hermes under high I/O load is that by design, Hermes can enable or disable different interrupt sources depending on which guest is active. In this test, we disabled the network port exception when the bare-metal serial port guest was running. This makes it impossible for the network port ISR to interrupt the user-mode code that handles the serial port receive. Operating systems in general do not support changing processor state for different threads[4], presumably because I/O transactions are assumed to be the domain of the operating system and mostly independent of user-mode software. That assumption was generally valid for early PCs and servers, whose job was primarily batch-mode processing with very little user interaction. Mobile and IoT devices have completely different set of requirements: they need to serve as a responsive user interface in which software works closely with I/O.

Uncertainty in scheduling can create real problems for these kinds of systems. For instance, if the same timing uncertainty in Figure 5.4 were imposed

[4]For example, we are not aware of any RTOS that allows the programmer to enable or disable different drivers while certain threads are running.

on ADC sampling in an IoT device, it could cause several decibels of harmonic distortion [24]. It is easy to imagine many situations in which timing errors could result in degraded system performance on mobile platforms.

The results in Figure 5.4 are an improvement to [68] which had an incomplete implementation of guest context switching in response to interrupts. At boot, we initialize an array of data structures containing (1) an interrupt priority and (2) a pointer to a guest that owns the interrupt. The interrupt number is implied by the position in the array. We also configure the ARM's interrupt controller with the same interrupt priorities as the array. When an interrupt occurs, it will be masked by the interrupt controller unless the `BASEPRI` register has a smaller mask value than the interrupt's priority. Interrupts with low priority are associated with less time-critical tasks.

If the interrupt has a higher priority than the `BASEPRI`, its priority is inserted into the `BASEPRI` so its handler cannot be preempted by lower-priority interrupts. The associated guest is then given control of the CPU, and the guest's ISR is executed. This method is a generalization of the one presented in [68], but it is more flexible for multiple exceptions.

In this method, when a high-priority exception occurs, the corresponding guest will immediately be given access to the CPU, regardless of what other guests or ISRs are currently executing. The user-mode code for that guest will have the opportunity to respond to the interrupt without waiting for other lower priority exceptions to finish.

The main goal of the Hermes hypervisor is to provide a thinner layer between hardware and software than is possible with an RTOS. There are three general techniques for virtualizing I/O:

Mobile Device Use Case

To demonstrate how Hermes would behave in a real deployment scenario, we built a prototype handheld device based on the SAME70 Xplained development board, and LCD touchscreen, and a GPS receiver, and camera. Our demonstration platform is meant to mimic a smart watch or other low-power mobile device.

Using the GPS receiver and LCD screen, we developed an app that tracks the user's location in real time and computes the speed based on successive measurements. We developed a simple service to display the current speed on the LCD and display a moving sprite-based background image. The moving background image is intended to mimic the user interface (such as a moving map) that such an app might display. We also implemented an app that handles network services to mimic a WiFi or Bluetooth interface that may be needed to send or receive application data (emails, map updates, etc.).

We also developed a video app that gathers frames from the video camera, postprocesses each frame in real time, and displays the frame to the user. Our postprocessing consists only of simple color correction since the embedded CPU is not capable of running sophisticated workloads like face recognition in real time. Still, the real-time color correction is near the limit of the CPU's capacity, and we found that additional workload such as a large volume of incoming network traffic put the system over its capacity, creating performance problems in an RTOS environment.

We ported these apps to the FreeRTOS and Hermes runtime environments to compare their performance. We focus on evaluating how real-time data from the GPS module is handled in both environments and how I/O latencies can affect the GPS apps can affect computation of instantaneous speed in the presence of a realistic mobile device workload.

Video App The board we use has a VGA image sensor interface that can accept images in 16-bit RGB color format at a rate of about 10 frames per second. Our video camera app has three major components: (1) a VGA interrupt service routine, (2) image data processing userland code (3) display ISR. All components are connected by 12-frame queues. Since the image processing code is a CPU hog, we do not want to run it in the VGA ISR. However, it must be run at high priority since it is in the critical path for the user interface. During the time that the video app is open—which we imagine would be relatively infrequent in our mobile device—the camera should be very responsive. The video app is a CPU-intensive real-time app that runs near our device's compute capacity.

	Avg Frame Loss Rate	Inter-Frame Jitter (σ)
FreeRTOS, Low I/O Load	0 frames/sec	7.03 ms
FreeRTOS, High I/O Load	4.2 FPS	47.91 ms
Hermes, Low I/O Load	0 FPS	4.85 ms
Hermes, High I/O Load	0 FPS	4.86 ms

Table 5.3: Performance comparison of Hermes and FreeRTOS in the video app. Low frame loss rate and jitter is better.

We implemented the app inside FreeRTOS and Hermes. In each runtime environment, we tested the app in two scenarios—under low I/O load and under a ping flood to simulate a high volume of network traffic. In the FreeRTOS implementation, we were not able to use the standard queue API to pass data between user code and interrupt code because high-priority ISRs are prohibited from making API calls. Instead, we implemented a custom queue outside of the FreeRTOS API. Normally the RTOS would schedule the userland image processing thread if there was data in the queue. But since our queue was outside the RTOS, we could not just block the task pending incoming data. Instead, we put the task to sleep for a short amount of time and poll the queue periodically. If the queue fills up rapidly before the userland code can poll, it can fill up and drop a lot of frames. This happens in Figure 5.5. To allow the video app to catch up, we had to manually kill the ping flood and allow the userland code to catch up.

We used the same queue in the Hermes implementation without frame loss. In the Hermes implementation, when the VGA ISR starts, the whole video app takes control of the CPU, including the userland code. As soon as the ISR returns, the userland code begins reading data from the queue without interruption from the networking app. In Hermes, related ISRs and user code are logically grouped together into virtual machines that run as independent units.

We used frame jitter and loss rate as metrics of performance. In both cases, we saw no frame losses under low I/O load. Figure 5.5 shows the number of frames dropped every second during a 276-second test of the FreeRTOS video app. **Under high I/O load, Hermes lost no frames, and**

the jitter did not increase because the hypervisor prioritizes userland video frame processing over lower-priority networking interrupts. In fact, our video processing software represents a relatively mild test case—we were able to cause much more dramatic frame jitter and loss rate in FreeRTOS by increasing the complexity of the video processing algorithm. Table 5.3 shows a performance comparison between Hermes and FreeRTOS for the video app under low and high I/O load conditions. Figure 5.7 shows histograms of the inter-frame timing for the same test cases.

The real-time app performs much better in the Hermes runtime environment than in the RTOS, but we can't get something for nothing. The interesting feature of the video app is that it has real time requirements and generates high CPU load. When Hermes prioritizes the video app, it starves the networking app of CPU time, resulting in extremely high (97%) packet loss rates for the ping flood (lower-speed pings were still lossless). Hermes gives programmers a knob to turn to trade real-time performance for fairness. We can dynamically adjust the VM priorities in real time to achieve the desired middle ground at run time.

GPS App The GPS module generates location estimates approximately once per second. To compute speed, the GPS app computes the distance between successive GPS coordinates and divides by one second. In our app, we only use the GPS to generate location estimates. We do not rely on its clock in our speed calculation. Instead, we use a clock internal to the ARM microcontroller to keep track of time. We do not want to treat the GPS module as an independent off-chip coprocessor. One of the goals of Hermes is to allow us to put as much of the computational load as possible on a single centralized CPU rather than dispatching sub-tasks (like speed computations) to single-purpose off-chip devices.

In the speed computation, there are three main sources of error: GPS location error, cruise control error, and I/O latency errors in the ARM CPU. We model each of these as additive white Gaussian noise. Since we did not have access to a digital readout of the car's speedometer, we did not have access to a reliable ground truth to compare our results. Instead, we will compare the variance of speed estimates.

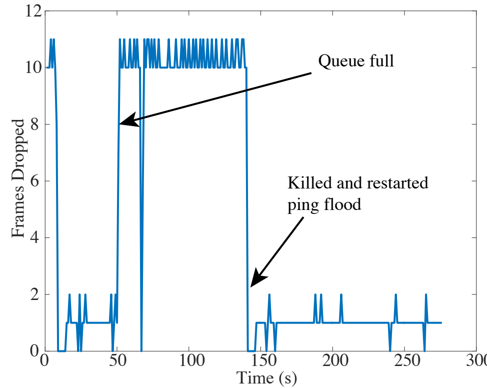

Figure 5.5: Number of frames dropped by FreeRTOS per second as a function of time while camera was running in the presence of a ping flood.

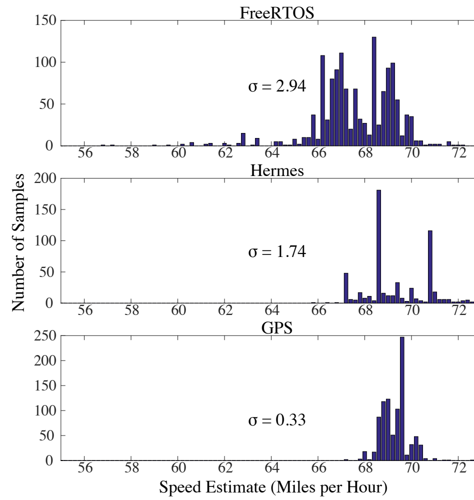

Figure 5.6: Histograms of speed estimates by our GPS app.

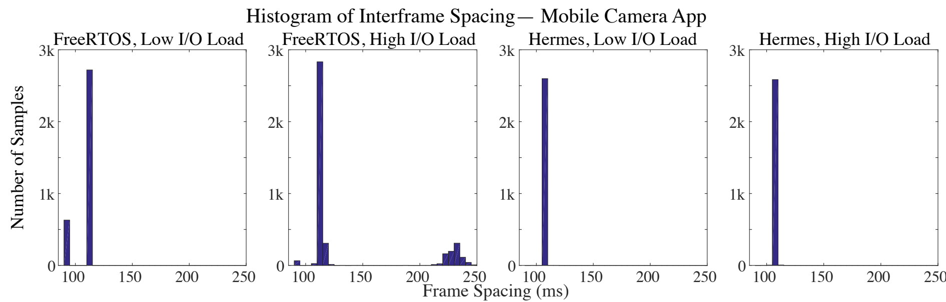

Figure 5.7: Histograms of inter-frame spacing for the video app. Standard deviation of these histograms are *jitter* in Table 5.3.

To benchmark the two different runtime environments, we set our mobile device up in a car and drove down a flat, straight stretch of highway with the cruise control set to about 67 miles per hour. Our mobile app's speed computations were logged to a file via the board's serial port. Histograms of our app's speed computations in the FreeRTOS and Hermes runtime environments

are plotted in Figure 5.6. For comparison, we have also plotted speed estimates using the GPS's internal timebase.

We are mostly interested in the variance of speed estimates, which is caused by uncertainty in the I/O event arrival time for the GPS data and the internal clock. Like the problem validation in Section 5.3, uncertainty is caused by competing I/O events from other apps running on the system—in this case from a more realistic workload. Each of the trials show in Figure 5.6 was acquired separately, which resulted in a slightly different average speed for each trial.

The GPS-only plot tells us about how much of the speed error is coming from errors in the GPS and the cruise control. The remainder of the variance in the FreeRTOS and Hermes histograms comes from errors in I/O event synchronization and jitter in the ARM CPU. Hermes has about 46% less variance than FreeRTOS after controlling for errors from the GPS.

Modifying the FreeRTOS Task for Hermes API Calls need to be converted from FreeRTOS to Atmel Software Framework (ASF), which is a board support library of drivers for bare-metal programs. Different CPUs or boards will use different board support libraries. Our FreeRTOS implementation already was using the ASF library for most of the drivers since it does not provide chip-specific drivers. For the most part, ASF drivers could be used out of the box without modification.

One exception is `vTaskDelay`, which is equivalent to `sleep` in Linux. `vTaskDelay` causes the current task to become inactive for a period of time and allowing other tasks to run. It is tightly coupled to the OS's scheduler. Bare-metal apps—which run as guests in Hermes—use an ASF library function called `Wait` which puts the CPU into a low-power sleep state. When converting our app to run in Hermes, we need to change all calls from `vTaskDelay` to `Wait`. Also in cases where we modified ASF drivers to be compatible with FreeRTOS, we need to revert to the stock ASF driver.

No Emulation	0.1 ms
Partial Emulation	0.3 ms
Passthrough	1 ms

Table 5.4: Comparison of ping round trip times in Hermes for three Ethernet driver implementations on the ARM device.

No Emulation	1675 kb/sec
Passthrough	12 kb/sec

Table 5.5: Comparison of SD card write throughput for three driver implementations on the ARM device.

I/O Benchmarking

We evaluated three techniques for virtualizing I/O operations:

- **Passthrough** uses interrupt and DMA remapping to give guests direct access to hardware resources.
- **Partial emulation** implements a reduced-function virtual hardware device with a custom device driver for the guest.
- **Full emulation** implements full emulation of the physical hardware device, including the full complement of registers, FIFOs, etc available on the hardware.

In this work, we studied passthrough and partial emulation, using the board's Ethernet and SD card interfaces as target I/O devices. The network driver is convenient because it is easy to benchmark using ICMP echoes (pings), and it's easy to compare to other virtualization platforms. We've benchmarked multiple I/O interfaces to see if there are any differences in performance.

Ethernet Interface Benchmarking Table 5.4 shows a comparison of round trip times for three different Ethernet driver implementations. The bare metal implementation is the unmodified driver supplied by the chip manufacturer with no virtualization; it is our reference implementation.

The bridged implementation is a custom driver running in the guest. Ethernet device interrupts are handled by Hermes without being passed up to the guest. The hypervisor presents a virtualized network interface to the guest,

and they hypervisor calls the chip manufacturer's driver functions to send and receive packets. The bridged driver allows multiple guests to share the same network interface by multiplexing incoming packets to the guests based on MAC address.

In the passthrough implementation, the guest runs the manufacturer's driver in raw form, emulated by Hermes. Ethernet device interrupts are caught by Hermes and passed to the guest, so all exception handling code is done in guest mode. The Ethernet device is not shared among multiple guests in this configuration.

Surprisingly, we find that the bridged (hypervisor-assisted) Ethernet driver performs far better than the passthrough. Since the passthrough driver runs all driver code in guest mode, all privileged instructions must be emulated by Hermes. This causes a significant slowdown in packet handling because the Ethernet driver has to invalidate a lot of data cache lines each time a packet is received, which requires many privileged instructions and memory accesses. In the bridged driver, the majority of privileged memory accesses and privileged instructions are done by the hypervisor, so they don't need to be emulated.

SD Card Benchmarking Table 5.5 shows a comparison of IO throughput different SD card driver implementations. The reason for the poor passthrough driver performance is that in the SD card write benchmark, the bottleneck of the implementation is in transferring a large volume of data through the ARM device's DMA interface. Each time we perform a block write to the SD card, the DMA interface invalidates about 1k lines in the data cache. The cache line invalidation operation is privileged, so it must be emulated by the hypervisor. If we do not invalidate those data cache lines during the block write, we can get a 3x performance improvement[5]. There are several other privileged bulk data transfer operations in the SD card write benchmark that create similar slowdowns. By comparison, the Ethernet driver needs to only transfer a small amount of data to respond to an ICMP echo.

[5] Of course, by skipping the data cache invalidation, the correct data will not be written to the SD card, but that does not matter since we are just writing random numbers for the benchmark.

SysTick Module Emulation

The SysTick module is an unusual case of I/O virtualization that requires special treatment by the hypervisor. Its job is to supply a periodic interrupt so that preemptive multitasking operating systems can keep track of time and share the CPU fairly among tasks. The SysTick module is unlike most other peripherals in that it needs to be used by more than one guest simultaneously. Unlike a UART or SPI interface, we cannot allow a single guest to monopolize the SysTick module because all other guests would not be able to switch between tasks. Also unlike many other peripherals, communication between the SysTick module and the CPU is unidirectional—SysTick periodically sends interrupts to the CPU, but the CPU doesn't send any data back. This means that there is no possibility of data getting garbled as two guests try to send different messages for example across a UART.

How should we manage the SysTick module such that the guests can share it fairly and they do not know that they are running in a virtualized environment? Since the SysTick module is used for guest scheduling, our architecture for emulating the SysTick module is closely linked to the scheduler's architecture. Decisions about how to emulate SysTick will affect the scheduler. We considered several approaches to this problem:

Round Robin Each time a hardware SysTick event occurs, we pass it to a different guest, alternating in round-robin style. We maintain one list of guests in the system, and we alternate which guest gets the CPU every time a SysTick exception occurs. For example, on a system with three guests, each guest receives every third SysTick exception, and the guest is allowed to run for one SysTick period. Each time a SysTick event occurs, we switch to a different guest. This approach has the benefit of being simple—we do not have to manage complex guest state, we can just cycle between guests. We also don't have to think much about the state of the SysTick module. The hardware SysTick module can be configured once at boot and forgotten about, and we do not have to store separate virtual SysTick state for each guest. We used round robin

SysTick and scheduling in early versions of Hermes for simplicity, but we no longer use it because it has several drawbacks.

The main drawback of this approach is that it has a very rigid fairness policy for the guests. Guests can only be run during their allocated timeslot. If a guest has no work to do, it can yield to a different guest, but this would disrupt the round-robin cycling. When a guest yields near the end of its timeslot, what guest gets to run next? Does the next guest get to run only until the end of the current timeslot, or does it get to use the next full timeslot as well? Furthermore, if different guests need to be interrupted at different frequencies, how do we do that?

One Guest-One Tick When a hardware SysTick event occurs, we execute the SysTick exception handler for every guest, all in succession. This approach gives a bit more flexibility in scheduling: we can maintain linked lists for guests that are in different states, for example waiting and active. When a SysTick exception occurs, we can move all guests from the waiting list to the active list and execute everyone's SysTick ISR. This approach is more fair than round robin because we can allow guests to yield the CPU without penalizing other guests. When a guest yields the CPU (for example by executing a sleep instruction), it is moved from an active list to a sleeping list, and only the other guests on the active list will be allowed to run. We don't waste CPU cycles by forcing a guest to run if it has nothing to do. In the round-robin scheduler, fairness was highest when inactive guests burned CPU cycles by looping. In one guest-one tick, only the guests with some work to do are allowed to use the CPU. The others wait on an inactive list for the next exception.

The drawback of this approach is that it does not allow different guests to set different SysTick frequencies. We have not tested this architecture.

Full SysTick Emulation Hermes maintains a separate virtual SysTick state for each guest. The hardware SysTick module generates interrupts faster than needed by any of the guests. Each time a hardware SysTick event occurs, Hermes updates the emulated state for each guest, subtracting one from the

virtual count register. When a guest's virtual count register reaches zero, Hermes runs the interrupt service routine for that guest.

One drawback of this approach is that it incurs more overhead in the hypervisor because the hypervisor's SysTick exception handler has to run more frequently.

This is the most flexible architecture for the virtualized SysTick for several reasons. First, since the hardware SysTick module is generating interrupts faster than needed by any of the guests, we can switch between guests at a finer time granularity. This means CPU sharing will be more responsive because we can switch between guests at a finer time granularity [18]. Also, full emulation allows different guests to have different SysTick interrupt periods; the previous methods forced all guests to use the same SysTick settings.

Because this is the most flexible way to emulate the SysTick module, it is the one we use in Hermes.

5.4 SUMMARY OF HERMES

We have demonstrated that AWESOME can improve the responsiveness of real-time I/O operations by creating separate virtual execution environments for each task on a real-time system. We arrived at our implementation by thinking at a high level about what is required to make the time-critical software as responsive as possible. In particular, we found that traditional real-time operating systems had an important shortcoming: they always prioritize interrupt handlers above user-mode processing, a vestige of early time-sharing mainframes that has the unintended consequence of occasionally running parts of a low-priority task before higher priority tasks. No amount of reconfiguration in the hardware interrupt controller can make this problem go away.

Instead, AWESOME creates a fully-isolated virtual execution environment for each of its guests. The most notable difference between a hypervisor and and RTOS is that the hypervisor can create a consistent virtualized CPU state for each guest that can be safely modified by the guest without affecting the other guests. The hypervisor's virtual CPU state can include things like the configuration of the interrupt controller, peripherals, etc. By contrast, RTOSes

only independently track the program counter and stack pointer for each task. Except for those two registers, RTOS tasks cannot modify CPU state without affecting other tasks. Doing so could cause the entire system to crash.

Our experiments revealed that I/O performance can be degraded—sometimes a little, sometimes a lot—by emulation in AWESOME. The question of whether or not that reduced I/O performance is acceptable is largely application-dependent. In this work, we made a high-level evaluation of the performance implications for I/O devices, and we will leave a more detailed analysis for future work. Also left for future work is an analysis of how competition among multiple real-time tasks will affect determinism in event responses. In the mean time, we have made the source code for AWESOME freely available for anyone to download.

6 RELATED WORK

In this section we describe prior work in the space of building automation and resource management.

6.1 ENERGY MONITORING

Energy monitoring represents a diverse body of work since it can be approached from many different angles (ie. sensor systems, cyber-physical building controls, feedback techniques, system architectures). Commercial entities also contribute to energy monitoring with a variety of their own sensors and architectural paradigms.

Energy Monitoring Hardware

Other research tends to focus on energy monitoring at the building level without being able to break it down further [44]. In a few cases, an infrastructure has been built around using commercially available hardware, such as [25]. Several commercial entities provide a fully implemented monitoring solution [3, 5, 8, 10], however each deployment is isolated from every other and access to the proprietary API or building data is often difficult. Some research also combines energy monitoring methods like [77], which contains both plug and building level monitoring. The focus of that work was to show how to reduce the energy consumption of the metering hardware using different methodologies. The three-layer architecture used by Emonix is very common for sensor networks, and is implemented in many energy monitoring research such as [25, 44, 77].

In general, there are several reasonable ways to monitor the electricity consumption of a building. Perhaps the simplest approach would be to monitor the total energy drawn by the whole building at the *utility service entrance*. This approach requires only a single monitor and is used in several commercial products we are aware of [5, 8, 10]. However, this approach does not allow residents to see their unique contribution to the overall energy consumption of the building. This makes it difficult for researchers to relate the overall

consumption to each occupant. Using building level energy consumption would only allow for building or campus scale predictions and interpretations of said data [16, 76]. Also, we do not want to include electric demand of appliances that residents do not have direct control over such as HVAC systems or hot water heaters because the demand, implementation, and efficiency of such appliances varies from building to building. In other words, we want our measurements to reflect the behavior of individual residents as much as possible.

The second possibility is to monitor electric demand on a per-appliance level. Using this technique, each appliance would have to be individually metered. For instance, one could deploy commercial plug-level energy monitors such as KillAWatt[4] or Watts Up?[12]. These would require one sensor board for each wall outlet and require that residents plug in their appliances to such monitoring units only.

While we do not focus on energy monitoring at the plug-level, there are several noteworthy research efforts using such systems. A pass-through plug level monitoring system, ACme [62], is designed around wireless technology with a fully implemented IPv6/6LoWPAN networking stack. A very similar system is proposed in [31], however the authors focus on other sensors to aid in the energy accountability for each person as they use appliances in common space.

Energy Profiling

Monitoring at the branch circuit level can have many advantages over other alternatives. In [16], the authors compare different buildings using the "Watt/sq-ft" metric. While Watt/sq-ft is a reasonable comparison method for building-level energy monitoring, in some cases it may be unfair. This unfairness could be derived from the heterogeneous nature of buildings from both a physical(age, updates, and maintenance) and utilization standpoint. For instance, can a server farm in a computer science building be directly compared to lab space in a chemistry building? Perhaps several servers could shut down, and their loads distributed to other active servers. However a lab's ventilation system cannot shutdown to save energy while the lab is in use.

The ability to add other parametric analysis techniques to building comparisons could be a powerful tool. Taking advantage of systems like Emonix, or even others such as [25] or [77] to compare people on a more direct level could lead to interesting ways to reduce energy consumption. Finally, using more specialized analysis techniques could result is better intervention mechanisms, which may help influence behavior patterns, such as those discovered in [26, 89].

6.2 heating and cooling monitoring and optimization

Network-connected controllers for residential HVAC systems have been introduced in various forms. The Nest thermostat [14] and the Bay Web Thermostat [13] are both commercial examples of smart thermostats. Both of these devices use a single point of measurement and a single point of control for the HVAC system without providing fine-grained control on a room-by-room basis.

RoomZoner is an academic system that uses distributed sensors to adjust the temperatures of individual rooms in a single-family house [102]. In this work, the authors use damper actuators to selectively cut off airflow to rooms that have reached their desired setpoints. The authors found that dampers worked well for a relatively small single-story house. However, in our experiments we found that this approach did not scale well to larger multi-story buildings.

PreHeat [100] and the Smart Thermostat [82] use occupancy data to establish a thermostat schedule that adapts to building occupants' use of their space. Thermovote [51] uses a crowd-sourcing or participatory sensing approach to establish an optimal temperature setting in an occupied space. In our work, we assume that an optimal thermostat schedule is available, and we focus on finding ways to fit local room temperatures to that schedule.

6.3 building water treatment system optimization

Most existing work in the area of smart water systems focuses on trying to reduce water consumption in the context of human behavior. As such, much of the existing work deals with points of interaction between building

occupants and the water distribution system, such as faucets, showers, and toilets. Hydrosense [55], NAWMS [65], and Driblet [45] are systems for identifying the sources of water consumption in a home using a single point of monitoring with the goal of classifying human behavior based on water usage. Winkler et. al. [106] devise a system to optimize the flow of water for irrigating lawns.

Another theme is to optimize the distribution of hot water throughout a residence. Circulo [56] and Hot Water DJ [94] aim to reduce the energy and water wasted while waiting for hot water to arrive at a sink or shower in a home. The Energy-Water Nexus [64] studies the relationship between energy and water consumption in a small collection of commercial buildings. Ranjan et. al. [96] leverage the energy-water nexus concept in their system by tracking localized energy and water consumption patterns to improve the accuracy of assignment of water consumption to building occupants.

Our work differs from the existing body of knowledge in that it addresses issues related to water treatment systems rather than water distribution systems.

The electric power research community has a body of work dedicated to load forecasting, used primarily for the purposes of generation planning, fuel purchasing, etc. [60]. In general, the time horizons of interest to the electric grid community are on the order of days to weeks. Charytoniuk and Chen achieved accurate forecasts of grid demand for time horizons of several hours using artificial neural networks [29]. We are not aware, however, of work in the electric power or microgrid community that responds to load forecasts with automatic control actions, a feature that is central to our work.

There has been much work in the building automation community centered on detecting and predicting occupancy. Beltran et. al. used thermal sensors deployed in common spaces to infer occupancy [21]. Ranjan et. al. used fixture-level water consumption data to infer individual room occupancy within a building [97]. Others have used electric power readings from smart meters to infer occupancy as well [30, 63].

There has also been a large body of work focused on responding to occupancy inferences or predictions [20, 50, 61, 93, 100]. Most of this work has focused on controlling thermostats in response to occupancy inferences

or predictions. Our work differs from this category because it directly uses historical sensor readings to predict future resource utilization without first narrowing the data down to a binary occupancy indicator.

6.4 MICROCONTROLLER OPERATING SYSTEMS AND HYPERVISORS

Other authors have explored real-time schedulers in hypervisors, in particular for Linux running in Xen [57, 107, 108]. Nemesis [78], an operating system that provided soft real-time guarantees for video processing, was aimed to solve a similar problem under similar assumptions, but it did not assume hard deadlines. There has also been work done porting [39] and evaluating [38] the KVM hypervisor on the ARM Cortex-A core, which is an application processor with a full MMU. Since we published Hermes, other authors at Intel have begun exploring the use of hypervisors for IoT and mobile systems for x86 CPUs [79, 109]. Hermes was the first hypervisor to have been implemented on MMUless machines—even early hypervisors ran on machines with memory management units [37].

Pinto et. al. have also worked on hypervisors for microcontrollers [90, 91]. Their work uses special-purpose hardware to enable multiple guest environments. They use ARM TrustZone [92], which is a set of hardware extensions that provides two separate architectural states: one for the hypervisor, the other for the guest. Each architectural state consists of a separate register file, stack, memory, etc. Context switches in the Pinto hypervisor can be done by modifying the guest's hardware architectural state.

Hermes does not use TrustZone to enable virtualization. Instead, guests run in unprivileged CPU mode, and Hermes traps and emulates privileged operations. The advantages of Hermes's trap-and-emulate technique are that (i) it does not require special hardware support and (ii) it can provide software emulation of virtual hardware devices, making it possible for multiple guests to share a single physical hardware interface. The disadvantage of Hermes is that privileged operations are slower in Hermes than on a TrustZone-based hypervisor.

FreeRTOS [23] is a popular real-time operating system for microcontrollers that we have run as a guest within Hermes. It provides a priority-based scheduler and inter-task communication APIs.

7 CONCLUSIONS

7.1 LESSONS LEARNED

In this work, we set out to build flexible IoT systems that could as seamlessly as possible integrate with one another to share information and access to services. We found that the difficulties in achieving our goal were caused by the client-server model that is baked in to the cloud services that most IoT systems use. Even if we relax some of our assumptions by moving cloud services to the edge, silos have a tendency to persist because of the centralized way in which we store and access data.

We found that APIs for storing, sending, and manipulating data were most useful when they were accessible at the device level rather than in the cloud. But building APIs at a per-device level comes with its own set of tradeoffs. Securing devices against intrusion becomes a much more difficult problem because they cannot just be cordoned off by a firewall. Standardization of APIs also becomes a much more important issue because interoperability depends inter-device communication. With Hermes, we dealt with some of these issues by building a software runtime environment for IoT devices that can expose a standard set of APIs without explicit cooperation from the manufacturer. But how to deploy a hypervisor on a third-party IoT device remains an open question.

7.2 FUTURE WORK

The long-term goal of this line of work is to build a software suite that makes the techniques presented in this dissertation deployable on any third-party mobile and IoT platforms. The ideas discussed in this dissertation, culminating with the development of the Hermes hypervisor, assume that we have some ability to modify the platform's hardware or software. Emonix, Hot/Cold and AWESOME are all custom-built IoT platforms on which we could load custom software and define a custom API. The reality on the ground is that most users will not be able to install a custom software set on their devices because the process is too complicated and time-consuming. Future work will focus on

making Hermes and other virtualization techniques deployable on real mobile and IoT devices.

Hermes on FitBit

Our first aim is to demonstrate that the Hermes hypervisor can run on a production hardware platform. This is a challenging problem because in general the stock software that runs on a platform like FitBit is closed source. The only way we can gain access to the software is by downloading the binary off of the microcontroller and disassembling it. We would then need to make the stock software—FitBit OS for example—run as a guest inside Hermes. This involves modifying the original binary, replacing some privileged instructions and memory accesses with traps to the Hermes hypervisor.

When we built Hermes, we had access to the source code for the experimental guests. In our experiments we mostly used FreeRTOS, the most popular open source real time operating system, as a guest. Some of the source code for our experimental guests needed to be modified either to improve driver performance or to add workarounds for the lack of virtualization support in the ARM microcontroller. As we try to run closed source third-party software as a guest inside Hermes, we will not have the luxury of significantly modifying the software at the source code level. We may be able to change a few instructions in the binary to add workarounds for hardware support, but we would not be able to rewrite hardware drivers for performance optimization. This will be challenging.

One of the important questions of this line of work is whether the performance penalty of using a hypervisor acceptable. It is possible that delays incurred by emulating privileged operations would slow down the user interface to the point of being unusable.

Another question is how to make it easy for users to load the Hermes hypervisor onto their devices. In our early experiments with loading a modified binary image to the FitBIt, we have disassembled the device to gain access to the circuit board and use a specialized in-circuit programmer to load new software. These techniques will not be available to most users. Classen et al. [53] have

demonstrated a technique for loading rogue binaries onto some models of FitBit using an over-the-air downloads. FitBit considers heir methods to be exploits, and FitBit OS has since been patched to prevent them.

APoGEE

In situations where installing a custom operating system is not an option, we want to develop techniques for sandboxing individual apps. The goal is to make a solution that is easily deployable by average users. Determined and knowledgeable users have the option of rooting their mobile devices to gain access needed for installing the virtualization software. But device rooting is too cumbersome for most users, and we can't expect that virtualization technologies that depend on it will be widely adopted by users. It is unfortunate that users cannot install virtualization platforms on their phones in the same way they install apps.

One practical technique for deploying virtualization technology on third-party devices is based on a legacy Linux kernel feature called User Mode Linux. UML is an early port of the Linux kernel that allows it to be run as a user mode process inside a host Linux machine [42, 43]. This method allows users to virtualize multiple operating systems on Android-based mobile devices without bootloader or root access. The only requirement is the ability to download and run custom software, as is typically allowed on stock Android distributions.

User-mode Linux is a set of modifications to the mainline Linux kernel that dates to the late 1990s, before the widespread availability of PC hypervisors. It was originally built to allow developers to quickly test modifications of the Linux kernel without rebooting into an experimental kernel. UML allows an unprivileged (non-root) user to boot the Linux kernel using a root filesystem image running as a userland process inside a host Linux system. Privileged operations such as I/O access, page faults, etc. are trapped by the host and dispatched to the UML process's signal handlers, which emulate the privileged operation in userspace.

Using UML as a sandbox, we could build in custom APIs at the kernel level to communicate with the third-party app. The main challenges of this work are

to make a bootable Android kernel and root file system that support most of the same features of the stock Android distribution. This would need to include a set of drivers that would allow the app to interact with the host operating system. In addition to providing a comprehensive set of features for the guest app, the UML root filesystem would need to be small enough to fit in memory and on the device's flash storage.

7.3 RELEVANCE TO TRENDS IN IOT

As the Internet of Things continues to evolve, we expect to see connected devices become more pervasive. To achieve this vision, the IoT will need to be accessible to more users, and the value that it provides will need to be clearer. Devices will need to be more interoperable and user interfaces will need to be more intuitive.

Migration of Cloud Services to the Edge

Researchers have recently been exploring the idea of distributing cloud services from centralized data centers to low-power computers like WiFi routers in homes. The goal is to reduce round-trip network latency and management costs for service providers. WiFi routers are a convenient place to host services because they often have underutilized CPUs. Edge services have not been widely adopted by mobile and IoT platforms, perhaps because there is not a standard platform for deploying software.

Expansion of IoT User Base

One of the biggest hurdles that the Internet of Things faces at present is the difficulty of installing and using devices. This lack of usability is reminiscent of the phenomenon of "computer illiteracy" of the 1990s, in which inexperienced users found it difficult to understand the dynamics of early PC interfaces. The difference is that early computers, despite their shortcomings, were essential tools that considerably increased productivity, and workers of the time

conformed to the clunky interfaces out of necessity. The crude user experience of modern IoT devices is not accompanied by the same indispensable functionality.

The IoT research community has recently made an effort to ease deployment and management of IoT devices. Context-based authentication and encryption [80, 85, 98] is an important step to make configuration transparent to the user. But once deployed and operating properly, IoT devices still require too much user interaction for the average person. Users are normally expected to interact with an app to directly control each device. As the density of interconnected devices increases, these interactions will consume more and more of our mental bandwidth. Although it was conceived to make life easier for users, the Internet of Things is on a trajectory to make simple interactions more complicated. The goal of this dissertation is to build a framework that allows IoT devices to interoperate, permitting a more wide-scale cooperation without user intervention.

Public Demand for Privacy

As the Internet of Things expands to collect more data about our surroundings, we are enabling more services, and interoperability offers more transparent cooperation among devices. But widespread and indiscriminate data collection threaten user privacy. Our goal should be to contour the data collection policies of IoT devices to store and transmit only information that is necessary for the operation. But apps often collect information from users without their knowledge and do not permit users to define fine-grained privacy policies. Hermes and APoGEE take aim at this problem by intercepting data at the operating system level before it can reach the putatively untrustworthy app. We can then filter the collected information in transit to make it compliant with the user privacy policy.

7.4 CONCLUDING REMARKS

This work is a first step toward building a cooperative framework for devices in the Internet of Things. We have demonstrated that device-level APIs can

improve interoperability of IoT services, and we have suggested some ways of building them in to systems. But more work is still needed to understand how to best deploy the techniques in this dissertation on new and existing systems. As the Internet of Things grows to a point where devices outnumber humans, we will need to think carefully about deployability, security, and upgradability of new and existing systems.

[1] Air distribution system installation and sealing: Proper duct installation increases efficiency, 2003. US Department of Energy Building Technologies Office.

[2] US Department of Energy, 2012. http://buildingsdatabook.eren.doe.gov.

[3] eGauge, 2013. http://www.egauge.net/products.php.

[4] Kill A Watt, 2013. http://www.p3international.com/products/special/p4400/.

[5] Lucid Design Group, 2013. http://www.luciddesigngroup.com.

[6] Madison Gas and Electric, 2013. http://www.mge.com/home/rates/tou.

[7] Powerhouse Dynamics, 2013. http://www.powerhousedynamics.com/buy-emonitor-dealers/.

[8] TED 5000 Home Electricity Monitor Store, 2013. http://www.theenergydetective.com/residential/store.html.

[9] The Modlet, 2013. http://themodlet.com/buy_office.html.

[10] Uihlein Electric, 2013. http://www.uihleinelectric.com/.

[11] Veris Inudsties, 2013. http://www.veris.com/Item/E30A042.aspx.

[12] Watts Up?, 2013. https://www.wattsupmeters.com/secure/products.php?pn=0.

[13] Bay Web Thermostat, 2014. http://www.bayweb.com.

[14] Nest, 2014. http://www.nest.com.

[15] Central arizona salinity study. Technical report, US Bureau of Reclamation, June 2015. http://www.usbr.gov/lc/phoenix/programs/cass/cass.html.

[16] Y. Agarwal, T. Weng, and R. K. Gupta. The energy dashboard: improving the visibility of energy consumption at a campus-wide scale. In *Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, BuildSys '09, pages 55–60, Berkeley, CA, USA, 2009. ACM.

[17] American Physical Society. Energy Units, 2014. http://www.aps.org/policy/reports/popa-reports/energy/units.cfm.

[18] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating systems: Three easy pieces*, volume 151. Arpaci-Dusseau Books Wisconsin, 2014.

[19] B. Balaji, J. Xu, A. Nwokafor, R. Gupta, and Y. Agarwal. Sentinel: Occupancy based hvac actuation using existing wifi infrastructure within commercial buildings. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 17:1–17:14, New York, NY, USA, 2013. ACM.

[20] N. Batra, A. Singh, and K. Whitehouse. If you measure it, can you improve it? exploring the value of energy disaggregation. In *Proceedings of the 2Nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*, BuildSys '15, pages 191–200, New York, NY, USA, 2015. ACM.

[21] A. Beltran, V. L. Erickson, and A. E. Cerpa. Thermosense: Occupancy thermal based sensing for hvac control. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings*, BuildSys'13, pages 11:1–11:8, New York, NY, USA, 2013. ACM.

[22] S. Berger, R. Cáceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. vtpm: Virtualizing the trusted platform module. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[23] R. Berry. FreeRTOS, 2017. http://www.freertos.org.

[24] B. Brannon and A. Barlow. Aperture uncertainty and adc system performance. *Application Note AN501*, 2006.

[25] R. Brewer and P. Johnson. Wattdepot: An open source software ecosystem for enterprise-scale energy data collection, storage, analysis, and visualization. In *Smart Grid Communications (SmartGridComm), 2010 First IEEE International Conference on*, pages 91–95, 2010.

[26] R. Brewer, G. Lee, and P. Johnson. The kukui cup: A dorm energy competition focused on sustainable behavior change and energy literacy. In *System Sciences (HICSS), 2011 44th Hawaii International Conference on*, pages 1–10, 2011.

[27] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente. On the difficulty of software-based attestation of embedded devices. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 400–409, New York, NY, USA, 2009. ACM.

[28] T. Chakraborty, A. U. Nambi, R. Chandra, R. Sharma, M. Swaminathan, Z. Kapetanovic, and J. Appavoo. Fall-curve: A novel primitive for iot fault detection and isolation. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, SenSys '18, pages 95–107, New York, NY, USA, 2018. ACM.

[29] W. Charytoniuk and M.-S. Chen. Very short-term load forecasting using artificial neural networks. *IEEE transactions on Power Systems*, 15(1):263–268, 2000.

[30] D. Chen, S. Barker, A. Subbaswamy, D. Irwin, and P. Shenoy. Non-intrusive occupancy monitoring using smart meters. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings*, BuildSys'13, pages 9:1–9:8, New York, NY, USA, 2013. ACM.

[31] Y. Cheng, K. Chen, B. Zhang, C.-J. M. Liang, X. Jiang, and F. Zhao. Accurate real-time occupant energy-footprinting in commercial buildings. In *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for*

Energy-Efficiency in Buildings, BuildSys '12, pages 115–122, Toronto, ON, Canada, 2012. ACM.

[32] Y. Cho, J. Shin, D. Kwon, M. Ham, Y. Kim, and Y. Paek. Hardware-assisted on-demand hypervisor activation for efficient security critical code execution on mobile devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 565–578, Denver, CO, 2016. USENIX Association.

[33] I. Corp. A guide to the internet of things. `https://www.intel.com/content/www/us/en/internet-of-things/infographics/guide-to-iot.html`, 2019.

[34] A. Corporation. Sam e arm cortex-m7 microcontrollers, 2017. http://www.atmel.com/products/microcontrollers/arm/sam-e.aspx.

[35] A. Corporation. Sam e70 xplained evaluation kit, 2017. http://www.atmel.com/tools/atsame70-xpld.aspx.

[36] A. Crabtree, T. Lodge, J. Colley, C. Greenhalgh, K. Glover, H. Haddadi, Y. Amar, R. Mortier, Q. Li, J. Moore, L. Wang, P. Yadav, J. Zhao, A. Brown, L. Urquhart, and D. McAuley. Building accountability into the internet of things: the iot databox model. *Journal of Reliable Intelligent Environments*, 4(1):39–55, Apr 2018.

[37] R. J. Creasy. The origin of the vm/370 time-sharing system. *IBM J. Res. Dev.*, 25(5):483–490, Sept. 1981.

[38] C. Dall, S.-W. Li, J. T. Lim, J. Nieh, and G. Koloventzos. Arm virtualization: Performance and architectural implications. *SIGARCH Comput. Archit. News*, 44(3):304–316, June 2016.

[39] C. Dall and J. Nieh. Kvm/arm: The design and implementation of the linux arm hypervisor. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 333–348, New York, NY, USA, 2014. ACM.

[40] N. Davies, N. Taft, M. Satyanarayanan, S. Clinch, and B. Amos. Privacy mediators: Helping IoT cross the chasm. In *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications*, HotMobile '16, pages 39–44, New York, NY, USA, 2016. ACM.

[41] U. S. Department of Energy. Residential Energy Consumption Survey (RECS), 2009. http://www.eia.gov/consumption/residential/data/2009/index.cfm.

[42] J. Dike. A user-mode port of the linux kernel. In *Annual Linux Showcase & Conference*. USENIX, 2000.

[43] J. Dike et al. User-mode linux. In *Annual Linux Showcase & Conference*. USENIX, 2001.

[44] M. Dominguez, J. J. Fuertes, S. Alonso, M. A. Prada, A. Moran, and P. Barrientos. Power monitoring system for university buildings: Architecture and advanced analysis tools. *Energy and Buildings*, 59(0):152 – 160, 2013.

[45] Driblet Labs. Driblet, 2015. http://driblet.co.

[46] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the ibm 4758 secure coprocessor. *Computer*, 34(10):57–66, Oct. 2001.

[47] R. Erickson and K. Lake. Third annual chloride progress report. Technical report, Madison Metropolitan Sewerage District, June 2013.

[48] V. L. Erickson, S. Achleitner, and A. E. Cerpa. Poem: Power-efficient occupancy-based energy management system. In *Proceedings of the 12th International Conference on Information Processing in Sensor Networks*, IPSN '13, pages 203–216, New York, NY, USA, 2013. ACM.

[49] V. L. Erickson, M. A. Carreira-Perpiñán, and A. E. Cerpa. Occupancy modeling and prediction for building energy management. *ACM Trans. Sen. Netw.*, 10(3):42:1–42:28, May 2014.

[50] V. L. Erickson, M. Á. Carreira-Perpiñán, and A. E. Cerpa. Observe: Occupancy-based system for efficient reduction of hvac energy. In *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, pages 258–269. IEEE, 2011.

[51] V. L. Erickson and A. E. Cerpa. Thermovote: Participatory sensing for efficient building HVAC conditioning. In *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, BuildSys '12, pages 9–16, New York, NY, USA, 2012. ACM.

[52] D. C. et. al. scikit-learn: Machine learning in python, 2015. http://scikit-learn.org.

[53] H. Fereidooni, J. Classen, T. Spink, P. Patras, M. Miettinen, A. Sadeghi, M. Hollick, and M. Conti. Breaking fitness records without moving: Reverse engineering and spoofing fitbit. *CoRR*, abs/1706.09165, 2017.

[54] M. Fomichev, F. Álvarez, D. Steinmetzer, P. Gardner-Stephen, and M. Hollick. Survey and systematization of secure device pairing. *IEEE Communications Surveys Tutorials*, 20(1):517–550, Firstquarter 2018.

[55] J. E. Froehlich, E. Larson, T. Campbell, C. Haggerty, J. Fogarty, and S. N. Patel. Hydrosense: Infrastructure-mediated single-point sensing of whole-home water activity. In *Proceedings of the 11th International Conference on Ubiquitous Computing*, Ubicomp '09, pages 235–244, New York, NY, USA, 2009. ACM.

[56] A. Frye, M. Goraczko, J. Liu, A. Prodhan, and K. Whitehouse. Circulo: Saving energy with just-in-time hot water recirculation. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings*, BuildSys'13, pages 16:1–16:8, New York, NY, USA, 2013. ACM.

[57] M. GarcÃƒÂ-a-Valls, T. Cucinotta, and C. Lu. Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture*, 60(9):726 – 740, 2014.

[58] Hach. Total hardness test kit, model 5-b, 2015. http://www.hach.com/total-hardness-test-kit-model-5-b/product?id=7640219508.

[59] D. C. Harris. *Exploring Chemical Analysis*. W. H. Freeman, 2012.

[60] L. Hernandez, C. Baladron, J. M. Aguiar, B. Carro, A. J. Sanchez-Esguevillas, J. Lloret, and J. Massana. A survey on electric power demand forecasting: future trends in smart grids, microgrids and smart buildings. *IEEE Communications Surveys & Tutorials*, 16(3):1460–1495, 2014.

[61] S. Iyengar, S. Kalra, A. Ghosh, D. Irwin, P. Shenoy, and B. Marlin. iprogram: Inferring smart schedules for dumb thermostats. In *Proceedings of the 2Nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*, BuildSys '15, pages 211–220, New York, NY, USA, 2015. ACM.

[62] X. Jiang, S. Dawson-Haggerty, P. Dutta, and D. Culler. Design and implementation of a high-fidelity ac metering network. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, IPSN '09, pages 253–264, Washington, DC, USA, 2009. IEEE Computer Society.

[63] M. Jin, R. Jia, Z. Kang, I. C. Konstantakopoulos, and C. J. Spanos. Presencesense: Zero-training algorithm for individual presence detection based on power monitoring. In *Proceedings of the 1st ACM Conference on Embedded Systems for Energy-Efficient Buildings*, BuildSys '14, pages 1–10, New York, NY, USA, 2014. ACM.

[64] J. Kadengal, S. Thirunavukkarasu, A. Vasan, V. Sarangan, and A. Sivasubramaniam. The energy-water nexus in campuses. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings*, BuildSys'13, pages 15:1–15:8, New York, NY, USA, 2013. ACM.

[65] Y. Kim, T. Schmid, Z. M. Charbiwala, J. Friedman, and M. B. Srivastava. Nawms: Nonintrusive autonomous water monitoring system. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, SenSys '08, pages 309–322, New York, NY, USA, 2008. ACM.

[66] W. Kleiminger, C. Beckel, and S. Santini. Household occupancy monitoring using electricity meters. In *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '15, pages 975–986, New York, NY, USA, 2015. ACM.

[67] N. Klingensmith. Building gcc as a cross-compiler, 2011. http://pages.cs.wisc.edu/~klingens/files/crossgcc.pdf.

[68] N. Klingensmith and S. Banerjee. Hermes: A real time hypervisor for mobile and iot systems. In *Proceedings of the 19th International Workshop on Mobile Computing Systems and Applications*, HotMobile '18, pages 1–6, New York, NY, USA, 2018. ACM.

[69] N. Klingensmith and S. Banerjee. Using virtualized task isolation to improve responsiveness in mobile and iot software. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, IoTDI '19, pages 160–171, New York, NY, USA, 2019. ACM.

[70] N. Klingensmith, J. Bomber, and S. Banerjee. Hot, cold and in between: Enabling fine-grained environmental control in homes for efficiency and comfort. In *Proceedings of the 5th International Conference on Future Energy Systems*, e-Energy '14, pages 123–132, New York, NY, USA, 2014. ACM.

[71] N. Klingensmith, Y. Kim, and S. Banerjee. A hypervisor-based privacy agent for mobile and iot systems. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, HotMobile '19, pages 21–26, New York, NY, USA, 2019. ACM.

[72] N. Klingensmith, A. Sridhar, Z. LaVallee, and S. Banerjee. Water or slime? a platform for automating water treatment systems: Poster abstract. In

Proceedings of the 2nd ACM Conference on Embedded Systems for Energy-Efficient Buildings, BuildSys '15, New York, NY, USA, 2015. ACM.

[73] N. Klingensmith, A. Sridhar, Z. LaVallee, and S. Banerjee. Spock: A sensor value prediction and online control algorithm for building resource management. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*, BuildSys '16, pages 123–132, New York, NY, USA, 2016. ACM.

[74] N. Klingensmith, D. Willis, and S. Banerjee. A distributed energy monitoring and analytics platform and its use cases. In *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings*, BuildSys'13, pages 36:1–36:2, New York, NY, USA, 2013. ACM.

[75] J. Z. Kolter and M. J. Johnson. Redd: A public data set for energy disaggregation research. *SustKDD Workshop on Data Mining Appliations in Sustainbility*, 2011.

[76] J. Z. Kolter and J. F. Jr. A large-scale study on predicting and contextualizing building energy use. *Proceedings of the Conference on Artificial Intelligence, Special Track on Computational Sustainability and AI*, 2011.

[77] D. Lachut, S. Piel, L. Choudhury, Y. Xiong, S. Rollins, K. Moran, and N. Banerjee. Minimizing intrusiveness in home energy measurement. In *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, BuildSys '12, pages 56–63, Toronto, ON, Canada, 2012. ACM.

[78] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE J.Sel. A. Commun.*, 14(7):1280–1297, Sept. 1996.

[79] H. Li, X. Xu, J. Ren, and Y. Dong. Acrn: A big little hypervisor for iot development. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS*

International Conference on Virtual Execution Environments, VEE 2019, pages 31–44, New York, NY, USA, 2019. ACM.

[80] Q. Lin, W. Xu, J. Liu, A. Khamis, W. Hu, M. Hassan, and A. Seneviratne. H2b: Heartbeat-based secret key generation using piezo vibration sensors. In *Proceedings of the 18th International Conference on Information Processing in Sensor Networks*, IPSN '19, pages 265–276, New York, NY, USA, 2019. ACM.

[81] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, Jan. 1973.

[82] J. Lu, T. Sookoor, V. Srinivasan, G. Gao, B. Holben, J. Stankovic, E. Field, and K. Whitehouse. The smart thermostat: Using occupancy sensors to save energy in homes. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 211–224, New York, NY, USA, 2010. ACM.

[83] Madison Gas and Electric. Degree day information, 2014. http://www.mge.com/customer-service/billing/about-bill/degree-day-info.htm.

[84] F. L. Mentec. Using the beaglebone pru to achieve realtime at low cost. *Embedded Related*, Apr. 2014. https://www.embeddedrelated.com/showarticle/586.php.

[85] M. Miettinen, N. Asokan, T. D. Nguyen, A.-R. Sadeghi, and M. Sobhani. Context-based zero-interaction pairing and key evolution for advanced personal devices. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 880–891, New York, NY, USA, 2014. ACM.

[86] C. Moratelli, S. Johann, and F. Hessel. Exploring embedded systems virtualization using mips virtualization module. In *Proceedings of the ACM International Conference on Computing Frontiers*, CF '16, pages 214–221, New York, NY, USA, 2016. ACM.

[87] A. V. Oppenheim and R. W. Schafer. *Discrete-time signal processing*. Pearson Higher Education, 2010.

[88] B. Parno, J. M. McCune, and A. Perrig. Bootstrapping trust in commodity computers. In *Security and privacy (SP), 2010 IEEE symposium on*, pages 414–429. IEEE, 2010.

[89] G. Peschiera and J. E. Taylor. The impact of peer network position on electricity consumption in building occupant networks utilizing energy feedback systems. *Energy and Buildings*, 2012.

[90] S. Pinto, H. Araujo, D. Oliveira, J. Martins, and A. Tavares. Virtualization on trustzone-enabled microcontrollers? voila! Real-Time and Embedded Technology and Applications Symposium, 4 2019.

[91] S. Pinto, J. Pereira, T. Gomes, M. Ekpanyapong, and A. Tavares. Towards a trustzone-assisted hypervisor for real-time embedded systems. *IEEE Computer Architecture Letters*, 16(2):158–161, 2016.

[92] S. Pinto and N. Santos. Demystifying arm trustzone: A comprehensive survey. *ACM Computing Surveys (CSUR)*, 51(6):130, 2019.

[93] D. Pisharoty, R. Yang, M. W. Newman, and K. Whitehouse. Thermocoach: Reducing home energy consumption with personalized thermostat recommendations. In *Proceedings of the 2Nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*, BuildSys '15, pages 201–210, New York, NY, USA, 2015. ACM.

[94] M. A. Prodhan and K. Whitehouse. Hot water dj: Saving energy by pre-mixing hot water. In *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, BuildSys '12, pages 91–98, New York, NY, USA, 2012. ACM.

[95] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten. ftpm: A software-only

implementation of a TPM chip. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 841–856, Austin, TX, 2016. USENIX Association.

[96] J. Ranjan, E. Griffiths, and K. Whitehouse. Discerning electrical and water usage by individuals in homes. In *Proceedings of the 1st ACM Conference on Embedded Systems for Energy-Efficient Buildings*, BuildSys '14, pages 20–29, New York, NY, USA, 2014. ACM.

[97] J. Ranjan, E. Griffiths, and K. Whitehouse. Discerning electrical and water usage by individuals in homes. In *Proceedings of the 1st ACM Conference on Embedded Systems for Energy-Efficient Buildings*, BuildSys '14, pages 20–29, New York, NY, USA, 2014. ACM.

[98] M. Rostami, A. Juels, and F. Koushanfar. Heart-to-heart (h2h): authentication for implanted medical devices. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 1099–1112, New York, NY, USA, 2013. ACM.

[99] F. Sant'Anna, N. Rodriguez, R. Ierusalimschy, O. Landsiedel, and P. Tsigas. Safe system-level concurrency on resource-constrained nodes. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 11:1–11:14, New York, NY, USA, 2013. ACM.

[100] J. Scott, A. Bernheim Brush, J. Krumm, B. Meyers, M. Hazas, S. Hodges, and N. Villar. Preheat: Controlling home heating using occupancy prediction. In *Proceedings of the 13th International Conference on Ubiquitous Computing*, UbiComp '11, pages 281–290, New York, NY, USA, 2011. ACM.

[101] P. Sommer and B. Kusy. Minerva: Distributed tracing and debugging in wireless sensor networks. In *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, SenSys '13, pages 12:1–12:14, New York, NY, USA, 2013. ACM.

[102] T. Sookoor and K. Whitehouse. RoomZoner: Occupancy-based room-level zoning of a centralized HVAC system. In *Proceedings of the ACM/IEEE*

4th International Conference on Cyber-Physical Systems, ICCPS '13, pages 209–218, New York, NY, USA, 2013. ACM.

[103] M. Tancreti, V. Sundaram, S. Bagchi, and P. Eugster. Tardis: Software-only system-level record and replay in wireless sensor networks. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, IPSN '15, pages 286–297, New York, NY, USA, 2015. ACM.

[104] I. S. Walker. Register closing effects on forced air heating system performance. Technical report, Lawrence Berkeley National Laboratory, January 2003.

[105] D. F. Willis, A. Dasgupta, and S. Banerjee. Paradrop: A multi-tenant platform for dynamically installed third party services on home gateways. In *Proceedings of the 2014 ACM SIGCOMM Workshop on Distributed Cloud Computing*, DCC '14, pages 43–44, New York, NY, USA, 2014. ACM.

[106] D. Winkler and A. E. Cerpa. Distributed independent actuation for irrigation control. In *Proceedings of the 1st ACM Conference on Embedded Systems for Energy-Efficient Buildings*, BuildSys '14, pages 148–151, New York, NY, USA, 2014. ACM.

[107] S. Xi, C. Li, C. Lu, C. D. Gill, M. Xu, L. T. Phan, I. Lee, and O. Sokolsky. Rt-open stack: Cpu resource management for real-time cloud computing. In *Cloud Computing (CLOUD), 2015 IEEE 8th International Conference on*, pages 179–186. IEEE, 2015.

[108] S. Xi, M. Xu, C. Lu, L. T. Phan, C. Gill, O. Sokolsky, and I. Lee. Real-time multi-core virtual machine scheduling in xen. In *Embedded Software (EMSOFT), 2014 International Conference on*, pages 1–10. IEEE, 2014.

[109] B. Yang, S. Yan, S. Liu, Z. Long, J. Yu, H. Zhang, Y. Yao, R. Xu, F. Feng, and J. Wu. Nezha: Mobile os virtualization framework for multiple clients on single computing platform. In *Proceedings of the 20th International Workshop on Mobile Computing Systems and Applications*, HotMobile '19, pages 39–44, New York, NY, USA, 2019. ACM.