

Application Development for Cyber-Physical Systems: Programming Language Concepts and Case Studies

A Dissertation

Presented to

the Faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Doctor of Philosophy (Computer Science)

by

Tamim I. Sookoor

August 2012

Abstract

Cyber-Physical Systems (CPSs) combine low-power radios with tiny embedded processors in order to simultaneously cover large geographic areas *and* provide high-resolution sensing/actuation. However, CPSs are extremely difficult to program. This issue is addressed by presenting a macro-programming framework called *MacroLab* that offers a vector programming abstraction similar to Matlab for Cyber-Physical Systems. The user writes a single program for the entire network using Matlab-like operations such as `addition`, `find`, and `max`. The framework executes these operations across the network in a distributed fashion, a centralized fashion, or something between the two – whichever is most efficient for the target deployment. This approach is called *deployment-specific code decomposition* (DSCD). MacroLab programs can be executed on mote-class hardware such as the Telos motes. The results indicate that MacroLab introduces almost no additional overhead in terms of message cost, power consumption, memory footprint, or CPU cycles over TinyOS programs.

As a crucial component of the application development cycle, debugging CPSs is addressed by *MDB*, the first system to support the debugging of macroprograms. MDB allows the user to set breakpoints and step through a macroprogram using a source-level debugging interface similar to GDB, a process called *macrodebugging*. A key challenge of MDB is to step through a macroprogram in sequential order even though it executes on the network in a distributed, asynchronous manner. Besides allowing the user to view distributed state, MDB also provides the ability to search for bugs over the entire history of distributed states. Finally, MDB allows the user to make hypothetical changes to a macroprogram and to see the effect on distributed state without the need to redeploy, execute, and test the new code. Macrodebugging is both easy and efficient: MDB consumes few system resources and requires few user commands to find the cause of bugs. A lightweight version of MDB, called *MDB Lite*, is also provided. It can be used during the deployment phase to reduce resource consumption while still eliminating the possibility of heisenbugs: changes in the manifestation

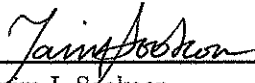
of bugs caused by enabling or disabling the debugger.

While a large number of CPS applications have been developed over the last few years, very few, if any, have been developed using a macroprogramming language. This is in spite of the vast array of macroprogramming languages, and abstractions, that are available. In an attempt to understand why macroprogramming languages are not being used for CPS application development and in order to evaluate the effectiveness of MacroLab in implementing a real-world CPS, a series of case studies involving the occupancy-based control of a Heating, Ventilation, and Air Conditioning (HVAC) system is implemented. Occupant-oriented HVAC control was selected due to it being a complete cyber-physical system involving the interaction between the real world and computation. Unlike many sensor network applications that involve primarily data collection, occupant-oriented HVAC control requires all three constituents of a complete CPS: sensing, computation, and actuation. The computation involved in these case studies range from making control decisions to predictions.

The case studies can be written in MacroLab in fewer lines of code than their implementations in Python and MacroLab can optimize their implementations depending on the topology of the network and capabilities of the sensors. Yet, the freedom for optimization decreases as the complexity of computation increases, or the size of the network decreases. Thus, for future cyber-physical systems the abstractions provided by macroprogramming systems, such as MacroLab presenting all sensor values as vectors and actuations as function calls, would be more beneficial than any implementation optimizations they may afford.

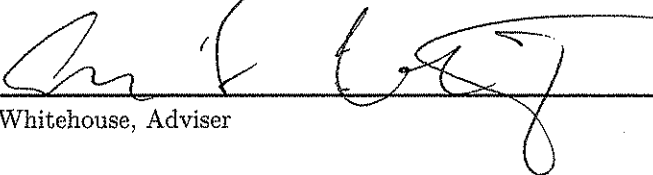
Approval Sheet

This dissertation is submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy (Computer Science)

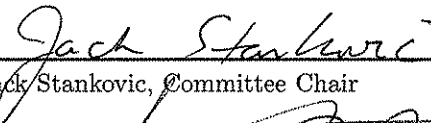


Tamim I. Sookoor

This dissertation has been read and approved by the Examining Committee:



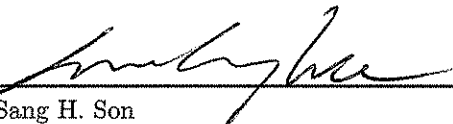
Kamin Whitehouse, Adviser



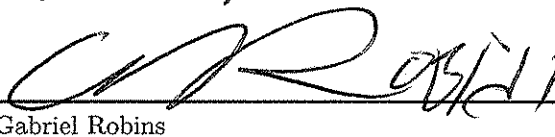
Jack Stankovic, Committee Chair



Stephen D. Patek

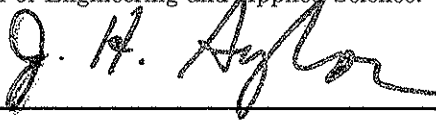


Sang H. Son



Gabriel Robins

Accepted for the School of Engineering and Applied Science:



James H. Aylor, Dean, School of Engineering and Applied Science

August 2012

Dedicated to my parents

Acknowledgements

I am in debt to the many great people who contributed to this work both directly and indirectly. Thanks to all of my collaborators who provided intellectual, technical, advisorial, and practical help with various aspects of this work. These include but are not limited to Timothy Hnat, Brian Holben, Jiakang Lu, Vijay Srinivasan, Virginia Smith and, of course, my dissertation committee.

Special thanks to Tasniya for her patience and constant support.

This work was funded in part by the National Defense Education Program Science, Mathematics & Research for Transformation Scholarship and the National Science Foundation Grant No. 1038271.

Tamim Sookoor

May 24, 2012

Charlottesville, VA

Contents

| | |
|--|-----------|
| Contents | vi |
| List of Tables | ix |
| List of Figures | x |
| 1 Introduction | 1 |
| 2 MacroLab | 6 |
| 2.1 Background and Related Work | 7 |
| 2.2 MacroLab Architecture | 9 |
| 2.2.1 Abstraction | 10 |
| 2.2.2 Cost Analyzer | 20 |
| 2.2.3 Compilation and Run-Time System | 23 |
| 2.3 Evaluation | 25 |
| 2.3.1 Expressiveness of the Abstraction | 25 |
| 2.3.2 Performance Overhead | 29 |
| 2.3.3 Effect of DSCD on Performance | 31 |
| 2.3.4 Accuracy of Static Cost Analyzer | 38 |
| 2.4 Conclusions | 39 |
| 3 MDB: The MacroLab Debugger | 41 |
| 3.1 Background and Related Work | 42 |
| 3.2 Three Types of Macroprogramming Bugs | 44 |
| 3.3 The MDB User Interface | 46 |
| 3.3.1 Logically Synchronous Views | 47 |
| 3.3.2 Temporally Synchronous Views | 50 |
| 3.3.3 Historical Search | 51 |
| 3.3.4 Hypothetical Changes | 53 |
| 3.4 The MDB Implementation | 56 |
| 3.4.1 Creating Execution Traces | 57 |
| 3.4.2 MDB Lite | 58 |
| 3.4.3 Distributed Timekeeping | 58 |
| 3.4.4 Generating Global Views | 59 |
| 3.4.5 Generating Hypothetical Changes | 60 |
| 3.5 Macroprogramming Properties for MDB | 62 |
| 3.6 Evaluation | 63 |
| 3.6.1 Experimental Setup | 63 |
| 3.6.2 Data vs. Event Logging | 64 |
| 3.6.3 RAM and Flash Overhead | 65 |
| 3.6.4 CPU Overhead | 66 |
| 3.6.5 Energy Consumption | 67 |

| | | |
|----------|---|------------|
| 3.6.6 | Discussion | 68 |
| 3.7 | Conclusions | 71 |
| 4 | Dual-Zone: Day/Night Zoning | 73 |
| 4.1 | Background and Related Work | 74 |
| 4.2 | Intuition and Preliminary Studies | 75 |
| 4.2.1 | Effect of an Oversized HVAC System | 75 |
| 4.2.2 | Inter-room Leakage | 76 |
| 4.2.3 | Room Occupancy | 77 |
| 4.3 | Implementation | 78 |
| 4.3.1 | Sensing House Temperature | 78 |
| 4.3.2 | Controlling Air-flow into Rooms | 79 |
| 4.3.3 | Controlling the HVAC System | 82 |
| 4.3.4 | Software Implementation | 83 |
| 4.4 | Evaluation | 87 |
| 4.4.1 | Zoning Evaluation | 87 |
| 4.4.2 | Macroprogramming Discussion | 88 |
| 4.5 | Conclusions | 91 |
| 5 | RoomZoner: Room-Level Zoning | 93 |
| 5.1 | Background and Related Work | 94 |
| 5.2 | Occupancy Assessment | 98 |
| 5.2.1 | Challenges | 100 |
| 5.2.2 | Approach | 101 |
| 5.3 | Zone Control | 106 |
| 5.3.1 | Goals | 107 |
| 5.3.2 | Challenges | 107 |
| 5.3.3 | Approach | 111 |
| 5.4 | Preliminary Implementations and Lessons Learned | 122 |
| 5.5 | Software Implementation | 125 |
| 5.6 | Evaluation | 126 |
| 5.6.1 | Zoning Evaluation | 127 |
| 5.6.2 | Macroprogramming Discussion | 129 |
| 5.7 | Conclusions | 130 |
| 6 | SmartZone: Predictive Zoning | 131 |
| 6.1 | Predictive Thermal Model | 131 |
| 6.1.1 | Introduction | 132 |
| 6.1.2 | Background and Related Work | 133 |
| 6.1.3 | Problem Definition | 133 |
| 6.1.4 | Experimental Setup | 135 |
| 6.1.5 | Model of Temperature Dynamics | 135 |
| 6.1.6 | Analysis | 137 |
| 6.1.7 | Results | 139 |
| 6.1.8 | Conclusions | 142 |
| 6.2 | Predictive Occupancy Model | 142 |
| 6.2.1 | Background and Related Work | 143 |
| 6.2.2 | Approach | 145 |
| 6.2.3 | Experimental Setup | 147 |
| 6.2.4 | Results | 148 |
| 6.2.5 | Sensitivity Analysis | 148 |

| | | |
|----------|---------------------------------------|------------|
| 6.2.6 | Conclusions | 154 |
| 6.3 | Predictive Control | 154 |
| 6.4 | Macroprogramming Discussion | 155 |
| 6.5 | Conclusions | 156 |
| 7 | Conclusion | 158 |
| | Bibliography | 162 |

List of Tables

| | | |
|-----|--|-----|
| 2.1 | Pseudocode for mote-level microcode translation of synchronous read | 17 |
| 2.2 | Pseudocode for mote-level microcode translation of synchronous write | 18 |
| 2.3 | Local half of naïve implementations of the synchronous max and find functions. . . . | 19 |
| 2.4 | Functions supported by the Run-Time System | 23 |
| 2.5 | Lines of code (LOC) comparison | 26 |
| 2.6 | MacroLab code size evaluation | 29 |
| 2.7 | MacroLab memory footprint | 30 |
| 2.8 | Execution time analysis | 31 |
| 3.1 | Basic commands provided by MDB | 47 |
| 3.2 | Flash memory consumption | 66 |
| 4.1 | Coarse-grained abstraction for thermostat operation | 82 |
| 5.1 | Threshold used to search for occupancy parameters | 105 |
| 5.2 | Conditioned Air Output for Different Heating and Cooling Stages | 110 |
| 5.3 | Area of, resistance of, and current through external surfaces. | 119 |
| 5.4 | MacroLab optimizations for case studies | 129 |
| 6.1 | Root Mean Square Errors | 138 |
| 6.2 | Example of a predictor with current time as feature. | 145 |
| 6.3 | Details of the 4 homes from which occupancy data was collected | 147 |

List of Figures

| | | |
|------|---|----|
| 2.1 | Example of a distributed macrovector | 12 |
| 2.2 | Example of dot-product indexing | 13 |
| 2.3 | Distributed Macrovector | 15 |
| 2.4 | Reflected Macrovector | 16 |
| 2.5 | MacroLab system architecture | 21 |
| 2.6 | A split-phase hardware driver for reading from the light sensor | 25 |
| 2.7 | A data collection application (Surge) in MacroLab | 27 |
| 2.8 | A tracking application (PEG) in MacroLab | 28 |
| 2.9 | Oscilloscope power measurements of MacroLab and nesC Surge implementations. . . | 31 |
| 2.10 | A bus tracking application in MacroLab | 32 |
| 2.11 | University Transit Service (UTS) bus routes at the University of Virginia (U.Va.) . . | 33 |
| 2.12 | Effect of deployment scenario on efficient implementation | 34 |
| 2.13 | Macrovectors in centralized and distributed implementations. | 35 |
| 2.14 | Effect of changing the base station range on code decomposition | 36 |
| 2.15 | Effect of adding a route to code decomposition | 37 |
| 2.16 | Estimated and measured messaging costs | 38 |
| 3.1 | MDB User Interface | 46 |
| 3.2 | Logically synchronous views | 49 |
| 3.3 | Example plot | 52 |
| 3.4 | Hypothetical Changes | 55 |
| 3.5 | Grandfather paradox | 60 |
| 3.6 | Evaluation testbed with 21 Tmote Sky motes | 63 |
| 3.7 | An acoustic monitoring application in MacroLab | 64 |
| 3.8 | Number of interrupts generated per 100 data states written to flash | 65 |
| 3.9 | RAM Overhead | 65 |
| 3.10 | CPU Overhead | 67 |
| 3.11 | Energy Consumption | 68 |
| 4.1 | Effect of Floorspace on Energy Usage | 76 |
| 4.2 | Effect of Exterior Walls on Energy Usage | 77 |
| 4.3 | Frequency of Room Usage Throughout a Day | 78 |
| 4.4 | Effect of Sensor Location on Temperature Reading | 79 |
| 4.5 | Three Generations of Active Registers and Bench-Top Test Rig | 80 |
| 4.6 | Details of an active register | 80 |
| 4.7 | Effect of Closing Registers on Air Flow | 81 |
| 4.8 | Finite State Machine of Dual-Zone Controller | 82 |
| 4.9 | The residential testbed used for zoning studies | 84 |
| 4.10 | MacroLab implementation of Dual-Zone. | 85 |

| | | |
|------|--|-----|
| 4.11 | MacroLab function to calculate transition in hysteresis state machine. | 86 |
| 4.12 | Energy usage of day/night zones vs. whole house conditioning | 87 |
| 4.13 | Effect of outdoor temperature on energy usage | 88 |
| 4.14 | Temperature Response of Conditioned and Unconditioned Rooms | 89 |
| 4.15 | Effect of Zoning on Airflow | 89 |
| 5.1 | Drawbacks of Programmable Thermostats | 95 |
| 5.2 | Drawbacks of Occupant-oriented Thermostats | 96 |
| 5.3 | Illustration of motion sensor data | 103 |
| 5.4 | Illustration of aggregated motion sensor data | 103 |
| 5.5 | Searching motion sensor data for room occupancies and vacancies | 104 |
| 5.6 | Transitional and Stable Occupancy Detected Using the Occupancy Model | 106 |
| 5.7 | Energy Efficiency and Lag Time for HVAC Heating Stages | 109 |
| 5.8 | Zone Control Data Flow Diagram | 111 |
| 5.9 | Android smartphone-based user interface to HVAC controller. | 112 |
| 5.10 | Finite-State Machine Used for HVAC Stage Selection. | 115 |
| 5.11 | Circuit model of house generated in Qucs circuit simulator. | 118 |
| 5.12 | MacroLab implementation of RoomZoner. | 126 |
| 5.13 | Energy usage of RoomZoner vs. whole house conditioning | 127 |
| 5.14 | Effect of outdoor temperature on energy usage for RoomZoner | 128 |
| 6.1 | Effect of HVAC system on temperature sensors | 134 |
| 6.2 | Example system parameters collected over one week | 135 |
| 6.3 | Predicted and measured temperatures | 137 |
| 6.4 | Aggregate plot of error distribution | 138 |
| 6.5 | Example of a temperature prediction | 140 |
| 6.6 | Error distributions for the static α model | 140 |
| 6.7 | Error distributions for the dynamic α model | 141 |
| 6.8 | Error distributions for the adjacency model | 141 |
| 6.9 | Illustration of a Percolator model instance | 146 |
| 6.10 | Illustration of observed feature set | 146 |
| 6.11 | Percolator Occupancy Model | 148 |
| 6.12 | Accuracy of Percolator as more training data is available | 149 |
| 6.13 | Accuracy of Percolator compared to its component predictors | 150 |
| 6.14 | Effect of training set size on predictor selection | 151 |
| 6.15 | Sensitivity of Percolator to the prediction horizon | 151 |
| 6.16 | Effect of increasing prediction horizon on occupancy predictors | 152 |
| 6.17 | Sensitivity of Percolator to the percolation threshold | 152 |
| 6.18 | Effect of percolation threshold on predictor selection | 153 |
| 6.19 | Effect of percolation depth on accuracy | 154 |
| 6.20 | MacroLab implementation of SmartZone. | 156 |

Chapter 1

Introduction

Cyber-Physical Systems (CPSs) combine low-power radios with tiny embedded processors in order to simultaneously cover large geographic areas *and* provide high-resolution sensing/actuation. This revolutionary technology has begun to deliver a new generation of engineering systems and scientific breakthroughs. However, CPSs are extremely difficult to program; building even a simple application entails several complex tasks such as distributed programming, resource management, and wireless networking. CPSs have reached a reasonable degree of technological maturity, but their impact and widespread adoption is limited by the complexity of their software.

Creating and debugging programs for CPSs is notoriously difficult. The most widely used programming paradigm for CPSs is node-level programming. *Node-level programming* is the process in which a developer manually creates the program that will run on each node, specifying node-local actions such as sensing, message passing, and local processing. These programs then execute on the nodes and the interactions between them produce emergent, network-wide behaviors. This programming model is notoriously difficult to use because the emergent network behavior is never explicitly specified and is instead fragmented among the programs of multiple different nodes. Furthermore, the emergent network behavior is difficult to predict: the user must have a mental model of each node and be able to mentally simulate the interactions between the nodes. This is particularly challenging given the complex, dynamic, and non-deterministic nature of CPSs: execution flows non-deterministically between nodes via unreliable broadcast messages and starts spontaneously on nodes due to timer and sensor interrupts. Despite these challenges, node-level programming is the most common way to program a WEN [1–3].

Macroprogramming is an emerging technology that aims to address this problem by providing high-level programming abstractions: the user writes a single *macroprogram* that specifies high-level distributed operations (i.e., leader election or contour finding), and the system automatically converts these into *microprograms* that specify local actions for each node (i.e., sensing, message passing, and local processing). Macroprograms do not actually execute on any node; all nodes execute microprograms, and the operations specified in the macroprogram are thus executed in a distributed fashion. Macroprogramming provides the user with the illusion of programming a single machine by abstracting away the low-level details of message passing and distributed computation. This promising approach has recently attracted dozens of prototype implementations with a wide array of programming models, including relational databases [4], geographic regions [5], logical rules [6], and data streams [7].

This dissertation presents the *MacroLab* framework for CPS software development and *MDB*, the first system to support the debugging of macroprograms. MacroLab is a macroprogramming framework that provides a vector programming abstraction using syntax similar to Matlab, which already has broad adoption among scientists and engineers. Data from sensors and actuators are manipulated just like other numerical vectors, making MacroLab programs similar to, and easy to integrate with, existing scientific software. Its traditional, imperative programming model supports general-purpose programming and is a natural way to encode CPS applications involving both sensing and actuation.

Since debugging is an important part of the software development process, the MacroLab Debugger (MDB) is implemented to aid users of MacroLab in CPS application development. MDB allows users to set breakpoints and step through macroprograms using traditional source-level debugging commands, much like GDB [8]. This provides the same abstraction as debugging a sequential program on a single machine, even though the macroprogram executes in a distributed, asynchronous manner on the network. This process is *macrodebugging*. A key challenge is to allow the user to step through a macroprogram in a sequential order, even if the nodes are not all executing the same distributed operations at any given time. MDB addresses this challenge by providing two ways to view distributed state: (1) *logical views* depict the distributed state where each node is executing the same logical operation in the macroprogram, although possibly at different times, and (2) *temporal views* depict distributed state of the entire system at a fixed time, even though nodes may not all be executing the same distributed operation. Both of these interfaces support *time travel*, which means

that the user can step both forward and backward through the code.

In addition to the ability to view distributed state, MDB provides two functions that are not supported by most existing source-level debuggers. First, *historical search* allows the user to search for the manifestation of a bug over the entire historical sequence of distributed states, without manually stepping forward and backward through the code. Second, MDB allows the user to make *hypothetical changes* to a macroprogram at debugging time, and to observe the effect of these changes on distributed network state without the need to redeploy, execute, and test the new code.

MDB fills an important gap in the macroprogramming tool chain, thereby making it easier to debug and deploy macroprograms. Furthermore, macrodebugging is both easier and has lower overhead than node-level debugging: the ability to view global, distributed state using the high-level macroprogramming abstractions eliminates the need to trace through the execution details of multiple nodes, such as message passing and hardware interrupts. Thus, MDB produces a marriage between debugging and macroprogramming that is mutually beneficial to both fields.

MDB is evaluated on three macroprograms running on a 21 node wireless testbed, and find that MDB has modest memory, execution, and energy overhead: approximately 300 B of memory, 0.5% of the CPU, and 30% energy overhead. This energy overhead is substantial enough that the user would probably disable MDB during the deployment phase, introducing the possibility of *heisenbugs*: changes in the manifestation of bugs caused by enabling or disabling the debugger. Therefore, MDB also includes a lightweight implementation called *MDB Lite* that only has 0.9% energy overhead. MDB Lite does not provide debugging support, but it does preserve the timing and memory characteristics of MDB, allowing the user to reduce energy overhead while still eliminating the possibility of heisenbugs.

While MacroLab theoretically provides an ideal framework for CPS application development, its effectiveness in the real world has not been evaluated. This is also true for many, if not most, other macroprogramming abstractions proposed for CPSs. Advanced CPSs still tend to use node-level programming with languages such as TinyOS that were developed over ten years ago for simpler wireless sensor networks. In order to understand the weaknesses of macroprogramming abstractions that have made their adoption so slow and to better understand the efficiency of MacroLab as a macroprogramming abstraction for CPSs, a number of case studies involving three versions of a smart home application are implemented. The application considered for this study is an occupant-oriented room-level heating, ventilation, and air conditioning (HVAC) zoning system called *Smart Zoning*.

Occupant-oriented HVAC control was selected because it is a compelling CPS application that allows the exploration of all aspects of cyber-physical systems. It involves sensing a physical environment using a heterogeneous collection of wirelessly connected sensors, data storage and computation for prediction and control, and actuation of physical hardware. Also, the system operates under real-time constraints since the actuations have to be performed in response to dynamic changes in the environment. Finally, the implementation of Smart Zoning involves the interaction between heterogeneous commercial off-the-shelf (COTS) sensors, a scenario that will be prevalent in many CPSs to come.

The HVAC controller evolves through three iterations from first controlling statically defined zones based on occupancy, to dynamically altering zones in response to changes in occupancy and temperature, to finally attempting to predict temperature responses and occupancy patterns. Throughout this evolution of the application, it is demonstrated that a program written in MacroLab can be changed relatively easily. MacroLab programs are also shown to provide varying levels of optimizations in terms of the actual program implementation as the program complexity grows. An insight gained from the case studies is that as the complexity of the programs grow, and/or the size of the network gets smaller, the optimizations afforded by a macroprogramming language, such as MacroLab, is limited.

This dissertation makes the following three contributions to the field of Cyber-Physical Systems:

MacroLab is a vector-based macroprogramming abstraction. MacroLab provides an easy-to-use programming abstraction based on the widely used Matlab language. MacroLab generates efficient binaries by optimizing applications based on user specified cost models and customizing the microprograms based on the network topology using deployment specific code decomposition.

MDB is the first debugger that allows debugging at the macroprogram source-level. MDB makes CPS application development easier by allowing developers to time-travel through an application and observe how its state changes over time, search for bugs over a historical sequence of distributed states, and see the effect of hypothetical changes on distributed state. All of these features are possible due to MDB being a post-mortem debugger that operates on data traces collected during system execution.

Occupant-oriented zoning was implemented in three phases with the first phase being Dual-Zone: a statically zoned system where a house is separated into two zones that are activated depending on the time of day. One zone is active during the day, while the other is active at night.

Dual-Zone uses 20.5% less energy than a traditional thermostat that conditioned the whole house as a single zone. The second phase was RoomZoner that enabled dynamically changing zones based on occupancy. RoomZoner was compared to the whole house being conditioned as a single zone but controlled with the same set of temperature sensors as RoomZoner instead of a traditional thermostat with a single temperature sensor. In this comparison, RoomZoner used 14.4% less energy than whole house conditioning. The third phase was SmartZone which is a predictive zoning system. While the predictive zoning controller was not implemented, two major components of such a system: a predictive temperature model and a predictive occupancy mode were implemented. The predictive occupancy model achieved over 75% accuracy across four houses with only 10 days of training data.

While these contributions advance the field of cyber-physical systems, there still remains a lot more work to be done to truly unleash the potential of the interaction of computers with the physical environment. While a world covered in sensors, as envisioned by the pioneers of wireless sensors networks a decade ago, has not yet arrived, people are surrounded by sensors in the form of smart phones and smart appliances that, if leveraged to build CPSs, could revolutionize the way they live. In order for this revolution to take place an easy way to program such a disparate collection of devices is essential. MacroLab would be ideal for such a scenario due to its extensibility to support new hardware platforms. Yet, its function libraries and collection of hardware drivers is greatly lacking for it to be useful in its present incarnation. Yet, releasing it as an open-source platform where developers can complete its function library and add to its collection of drivers can transform it into a tool that can propel the field of CPSs through the next decade and beyond.

Chapter 2

MacroLab

MacroLab is a macroprogramming abstraction that provides a vector-based syntax similar to Matlab [9]. All data on nodes, including sensor values, internal state, and parameters for actuation, are abstracted for the user as vectors called *macrovectors*. The user can operate on macrovectors with Matlab’s standard set of vector operations such as `max`, `min`, `sum`, or `find`¹, and the system compiles these down into local actions for each node that cooperatively execute the vector operations.

MacroLab introduces a new data structure called a *macrovector* which can be used to store in-network data such as sensor readings. Conceptually, each element of a macrovector corresponds to a different node in the network, but macrovectors can be stored in different ways. Each element can be on its corresponding node, all elements can be on a central server, or all elements can be replicated on all nodes. No matter how a macrovector is stored, it can support standard vector operations such as `addition`, `find`, and `max`. These operations may run in parallel on a distributed macrovector, sequentially on a centralized macrovector, or somewhere in between. Thus, by changing the representation of each macrovector, MacroLab can decompose a macroprogram in the way that is most efficient for a particular deployment. For example, it may use centralized representations for small star topologies and distributed representations in large mesh networks. This approach is called *deployment-specific code decomposition (DSCD)*.

In contrast to systems like TinyOS [10], the MacroLab programmer specifies application logic in terms of abstract computation and does not need to explicitly control data partitioning or message passing from within the source code. Instead, these tasks are performed automatically as compile-

¹Matlab’s `find` operator returns the indices of non-zero elements in a vector

time and run-time optimizations. By separating application logic from program decomposition, MacroLab can improve code portability, increase code reuse, and decrease overall development costs. Furthermore, it can reduce overall resource consumption. The results show that automatically choosing a decomposition for each deployment can reduce message passing by up to 50% over using a single decomposition for all deployments.

MacroLab provides a clear cost model so that the programmer can write code that produces efficient and optimized decompositions. This is analogous to the idea in Matlab that vectorized code is more efficient than `for` loops. MacroLab does not compromise on power or memory efficiency in order to provide a high-level vector programming abstraction and the results indicate that MacroLab programs are just as efficient as normal TinyOS programs.

2.1 Background and Related Work

Macroprogramming systems have been proposed with a wide variety of abstractions and programming models, each of which is designed to make programming easier for some class of applications. For example, database-like systems such as TinyDB [4] and Cougar [11] allow the user to specify the desired data using declarative SQL-like queries. These systems are most suitable for data collection applications where the desired data can be described with a declarative query. Several systems such as Hood [12], Regions [5], and Proto [13] are designed for spatial applications and allow users to specify operations over groups, neighborhoods, or regions in space. Other systems such as Semantic Streams [7], Flask [14], and Regiment [15] allow users to specify global operations in terms of *data streams* and *stream operators*. These are most suitable for defining a static set of long-running operations over streams of sensor data. Logical rule-based systems such as RuleCaster [16], DSN [17], and Snlog [6] allow the user to define a global objective in terms of system wide invariants that must be enforced at run-time. MacroLab is perhaps most similar to imperative macroprogramming abstractions like Marionette [18], Pleiades [19], Kairos [20], COSMOS [21], and Tenet [22]. These systems support general-purpose programming with a traditional imperative programming model. Vicaire et al. proposed the Bundles [23] programming abstraction that address some issues inherent in CPSs that were not tackled by other programming abstractions. These issues include the possibility of CPSs being systems of systems where the subsystems exist across administrative domains. Bundles provides an abstraction of a logical collection of devices that can be used by different

groups of people with different access restrictions. Bundles also handles mobility within and across CPSs. MacroLab is the first macroprogramming system for CPSs to provide vector programming, a powerful and concise abstraction that already has wide adoption among scientists and engineers.

Several existing systems allow users to write imperative programs that can then be distributed across multiple processors for the purposes of high performance computing. These include High Performance Fortran (HPF) [24], Fortran D [25], and Split-C [26]. The fundamental difference between these approaches and MacroLab is their dependence on the user to specify how the data and operations should be distributed. For example, Fortran D uses the statements `decomposition`, `align`, and `distribute` to specify how to execute a program on multiple processors. In contrast, MacroLab programs do not specify how to map the computation onto the network. In fact, the system will create a different mapping for each network on which the program is executed.

Other systems such as MagnetOS [27], Coign [28], and J-Orchestra [29] can automatically decompose a program and distribute it across a network in order to minimize network traffic. Similar to MacroLab, these systems use program profiling to tailor the decomposition to a specific network topology. In contrast to MacroLab, these systems decompose programs at the *object level*. MagnetOS and J-Orchestra break a program up at the boundaries of Java objects and use Java RMI between segments of the program. Coign requires programs to conform to Microsoft’s Component Object Model (COM) and breaks them up at the boundary of the COM objects. MacroLab introduces parallelism at the level of individual operations instead of at the level of objects or software components.

Finally, many systems allow the user to specify parallel operations using parallel data structures. SET Language (SETL) [30] provides primitive operations such as set membership, union, intersection, and power set construction, which can be applied in parallel to elements of *unordered sets*. Starlisp (*Lisp) can apply vector operations such as vector addition and multiplication over *Parallel Variables (PVARs)* which are vectors with one element per processor. Similarly, NESL allows parallel operations on *sequences*. These are similar to MacroLab’s parallel vector operations on *macrovectors*. However, MacroLab goes beyond these systems by employing *multiple* underlying representations of a macrovector. Unordered sets, PVARs, and sequences can only be decomposed in one way while macrovectors are decomposed in one of many different ways depending on the topology over which the program is executed. MacroLab is the first system that can perform automatic, topology-specific decomposition on programs describing parallel operations on parallel data

structures.

He et al. [31] present an architecture for wireless sensor networks called *Essentia* which advocates for asymmetric function placement. This approach recommends decoupling the non-essential functionality from the core of the sensor network and placing it at the basestation. This allows the highly resource constrained nodes to only perform essential tasks and rely on a more powerful basestation to perform more demanding activities. MacroLab advocates for applications to be developed without being constrained by such an architecture. Certain applications, in specific topologies, function much more efficiently when a large portion of the computation is pushed into the network. Yet, MacroLab does not burden the programmer with deciding how an application should be implemented. Instead it automatically decomposes a centralized program to be executed efficiently on the available resources.

2.2 MacroLab Architecture

MacroLab allows the user to write a single program that is simple, robust, and manageable and then automatically decomposes it depending on the target deployment. A *program decomposition* is a specification of where data is stored in the network and how messages are passed and computations are performed in order to execute the program. A macroprogram may be decomposed into *distributed* operations for a large mesh network, where data is stored on every node and network operations are performed in-network. It could also be decomposed into *centralized* operations for a small star topology, where all data is collected to a central base station. A program may also be decomposed into many points on the spectrum between purely centralized or purely distributed code. The implementation could also use group-based data processing or in-network aggregation.

MacroLab's overall architecture is depicted in Figure 2.5. A macroprogram (described in Section 2.2.1) is passed to the decomposer (Section 2.2.1) which generates multiple decompositions of the macroprogram. Each decomposition is passed to the cost analyzer (Section 2.2.2) which calculates the cost of each with respect to the cost profile of the target deployment. This cost profile must be provided by the user and may include information such as the topology, power restrictions, and descriptions of the radio hardware. The cost analyzer chooses the best decomposition and passes it to the compiler and run-time system (Section 2.2.3) which converts the decomposition into a binary executable and executes it. While it executes, the program and the run-time system continue to

collect information about the cost profile of the deployment and feed this information back to the cost analyzer. If the cost profile changes or if the cost profile at compile time was incomplete or incorrect, the cost analyzer may decide to reprogram the network with a new decomposition.

The architecture of MacroLab is presented in 2.5. An alternative implementation would be a compiler using the cost profile implementation to produce the correct decomposition the first time, but it is conceptually easier to enumerate all possible implementations and evaluate them.

The following subsections discuss the four components of MacroLab: the programming abstraction, the decomposer, the cost analyzer, and the run-time system.

2.2.1 Abstraction

MacroLab provides a *vector programming* abstraction similar to Matlab. A vector is a data structure containing values called *elements* that are arranged in rows and columns. For example,

$$r = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

is a 3 x 3 vector with 9 elements. Vectors can be indexed by dimension: the element in the second row and the third column of r can be selected with `r(2,3)` resulting in f . In MacroLab, like in Matlab, the “:” is used to select an entire dimension. For example, `r(3,:)` selects the 3rd element of the first dimension (rows) and the entire second dimension (columns), namely `[g h i]`. Operations such as *vector addition* and *vector multiplication* operate on the data structures. The operation `find(r==f)` produces the index of the element in r that has the value f , which is `[2,3]`. In addition to standard vector programming, MacroLab introduces three new concepts to facilitate software development for CPSs: the macrovector, the dot-product index, and the neighborhood. These concepts are discussed in the next three subsections.

The Macrovector

MacroLab introduces a new data structure called the *macrovector*. Macrovectors differ from traditional vectors in that each element of a macrovector is associated with a particular node in the network. Thus, macrovectors are *unordered* and are *indexed by node ID*. This abstraction can be

useful, for example, to store sensor readings. If `light` is a macrovector storing the light values of each sensor, then the operation `light(5)` would retrieve the light value of the sensor node with $ID = 5$. Since sensor node IDs may be non-sequential, the elements in a macrovector do not form a strict sequence. Macrovectors can have multiple dimensions, but only a single dimension is indexed by node ID. The other dimensions are normal vectors indexed sequentially. Macrovectors can be created using the command

```
light = Macrovector(<scope>, [length], [length], ...)
```

where the `scope` of the macrovector is the set of nodes with which the elements are associated. This scope is a vector of node IDs and the length of the first dimension will be the number of IDs. The lengths of subsequent dimensions must be given for a multi-dimensional macrovector. These `lengths` are simply integer values indicating the size of each dimension.

Macrovectors support many standard Matlab vector operations such as `addition`, `subtraction`, `cross-product`, `find`, `max`, and `min`. These operations can be combined to perform *macro* operations that operate on data associated with many different sensor nodes. For example, the operation

```
maxLight = max( light )
```

will return the maximum light value in the network. The operation

```
brightNodes = find( light > mean(light))
```

will return the IDs of nodes that have above-average light values. The operation

```
hotLight = light( find( temp > 100 ) )
```

will return the light values on nodes where the `temp` value is higher than 100. If these vectors were tables in TinyDB, this would be similar to posing the SQL query

```
SELECT light WHERE temp > 100
```

Elements associated with the same node ID are paired together for binary operations that involve multiple macrovectors. For example, Figure 2.1 shows the operation $C = A + B$ performed on three $n \times 2$ macrovectors. In this operation, the elements of A and B corresponding to node 35, for example, are added together and stored in the elements of C corresponding to node 35.

| C | | | A | | | B | | |
|----|----|----|----|---|----|----|---|---|
| 35 | 10 | 7 | 35 | 8 | 4 | 35 | 2 | 3 |
| 2 | 12 | 7 | 2 | 9 | 3 | 2 | 3 | 4 |
| 18 | 13 | 15 | 18 | 9 | 10 | 18 | 4 | 5 |
| 94 | 6 | 13 | 94 | 1 | 7 | 94 | 5 | 6 |
| 10 | 12 | 9 | 10 | 6 | 2 | 10 | 6 | 7 |
| 61 | 9 | 11 | 61 | 2 | 3 | 61 | 7 | 8 |

Figure 2.1: Two $n \times 2$ macrovectors A and B can be added and stored into a third macrovector C . The values on the left of each vector indicate which node ID the cells are associated with.

Dot-product Index

MacroLab provides a new way of indexing into macrovectors called the *dot-product index*. For example, with $\mathbf{s} = [1 \ 2 \ 3]$ and $\mathbf{t} = [2 \ 1 \ 3]$, the aforementioned vector \mathbf{r} can be indexed as follows:

$$r(s, t)[1, 2] == \begin{bmatrix} & b & \\ d & & \\ & & i \end{bmatrix}$$

The two values in square brackets indicate that the elements of the first and second dimension indices should be matched pair-wise before values are selected from the matrix. Since s and t each contain 3 elements, this dot-product index would select 3 elements from r in total. With the values of s and t above, the dot-product index would select elements $[1, 2]$, $[2, 1]$, and $[3, 3]$. This is different from traditional indexing in Matlab, which might be called the *cross-product index*, in which the same index vectors would produce

$$r(s, t) = \begin{bmatrix} b & a & c \\ e & d & f \\ h & g & i \end{bmatrix}$$

In other words, all values of index vector s are paired with all values of index vector t , selecting 9 values in total. Figure 2.2 illustrates how cross-products and dot-products can be used to select different elements in a three-dimensional vector.

Dot-product indexing can be used to efficiently perform operations in which the element to be selected on each node is different. For example, if an $n \times 10$ macrovector `circularBuffer` stored the

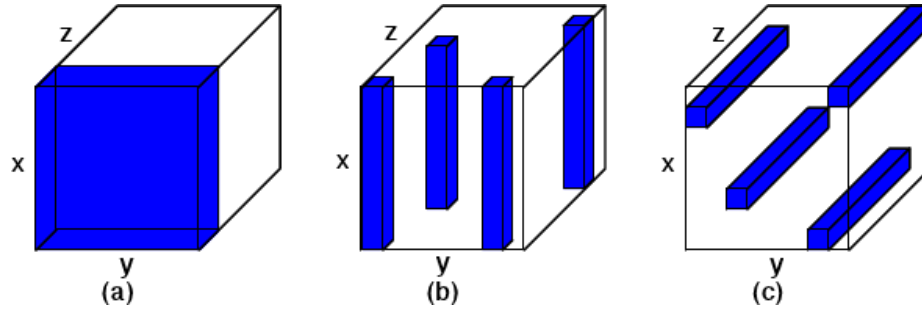


Figure 2.2: A three-dimensional vector D can be indexed with the (a) cross-product index $D(x, y, 1)$; (b) dot-product index $D(:, y, z)$ [2, 3]; or (c) dot-product index $D(x, y, :)$ [1, 2].

last 10 light values read on each node and an $n \times 1$ macrovector `lastIndex` stored the index of the most recent value stored, then the operation `circularBuffer(:, lastIndex) [1, 2]` would provide the most recent value on each node. The capabilities of macrovectors and dot-product indexing will be demonstrated in Section 2.3.

Neighbor-based Representation

A node's *neighborhood* is the set of nodes that are within radio range. This is a very useful type of *group* in which a node is guaranteed to have cheap communication to all other nodes. Connectivity-based neighborhoods are often a critical part of efficient in-network data processing algorithms. Neighborhoods are a special type of group since each node has a different neighborhood. Because of this, new syntax for defining a neighbor-based macrovector is introduced:

```
lightReflection = neighborReflection(light)
```

which indicates that `lightReflection` is a vector of neighbor-based macrovectors that should store *reflections* of the `light` macrovector. When a node writes to its own element of the `light` macrovector, that value is cached in the rows corresponding to its neighbors. Thus, `lightReflection` is a two dimensional vector where each row contains cached values of a node's neighbors' light readings. Since each node may have a neighborhood of different size, this is not necessarily a rectangular matrix; each row may be a different length. This abstraction is very similar to the Hood programming abstraction [12].

Program Decomposition

The MacroLab decomposer converts a macroprogram into a set of microprograms that can be executed on nodes. The goal is to preserve the semantics of the macroprogram while allowing for efficient distributed operation. The decomposition algorithm has two steps. First, it chooses a data *representation* for each macrovector, which can be *distributed*, *centralized*, or *reflected* (Section 2.2.1). Based on the representations chosen, it then uses rule-based translation to convert the vector operations in the macroprogram into network operations (Section 2.2.1).

Choosing Macrovector Representations

Macrovectors provide a uniform interface to several underlying *representations*, which are different ways that the macrovector can be stored in the network. MacroLab currently supports three representations: distributed, centralized, and reflected, the trade-offs of which are described in more detail below. Other representations are possible and would allow MacroLab to support different classes of distributed algorithms. Vector operations can be applied to macrovectors regardless of their representation, making them ideally suited for DSCD.

With three possible macrovector representations, a program with four macrovectors would have 3^4 possible decompositions. Thus, the space of decompositions grows exponentially with the number of macrovectors in a program. Currently, MacroLab must systematically explore all possible combinations of representations for the macrovectors in a program in order to find the optimal decomposition. This is currently not a problem for programs with only a small number of macrovectors. For the programs described in this section, the translation process from macrocode to microcode takes less than 0.5 seconds. In contrast, compilation of the microcode to a binary executable takes over 13 seconds, using the nesC and Matlab compilers.

Distributed Representation: The first way to represent a macrovector is to store each row on its associated node. Figure 2.3 shows how the elements of the macrovectors *A*, *B*, and *C* from Figure 2.1 can be stored on each node and how the **addition** operation is performed. Since elements are only added to corresponding elements on the same node, this operation can take place without message passing between nodes.

In general, the distributed representation of macrovectors allows for the efficient implementation of vector operations that do not span multiple rows. Conversely, this representation requires sig-

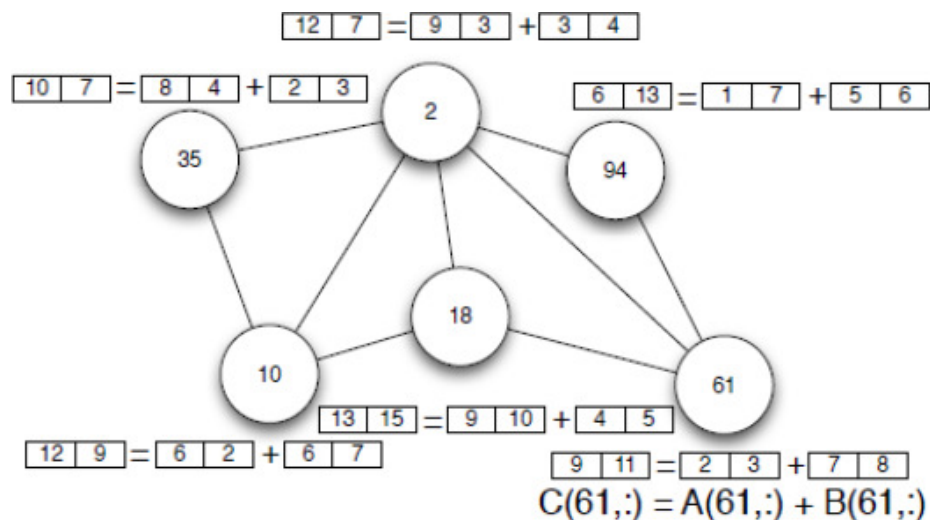


Figure 2.3: Distributed Macrovector: Nodes can read and write their own values in the vector.

nificant message passing for aggregate operations like **max** that require values resident on multiple nodes. If a macroprogram uses the **max** operation frequently on a particular macrovector, then a distributed decomposition would be very costly.

Centralized Representation: The second representation supported by MacroLab stores all elements on a single node, typically the base station. This representation is in diametric opposition to a *distributed* representation. It allows operations like **max** to be applied with virtually no explicit message passing cost. However, there is a potentially significant cost associated with keeping the elements of the centralized vector up-to-date. If the values are frequently updated remotely by the sensor nodes, they need to be frequently transmitted for storage. The centralized representation is favorable if the vector participates frequently in aggregate operations that span rows (like **max**). It is less favorable if the vector is frequently updated with sensor data.

Reflected Representation: The third macrovector representation stores all elements on all nodes. The microcode on each node has read/write access to its associated element and read-only access to cached versions of all other elements in the vector. This precludes the need for write-write synchronization since only one node may write to any given element. However, nodes do need to communicate their updated values after performing a write, as illustrated in Figure 2.4; when node 35 writes to its own element in the vector, the value is *reflected* to all other nodes currently caching it.

This representation is conceptually similar to Reflective Memory (RM), which is a form of shared

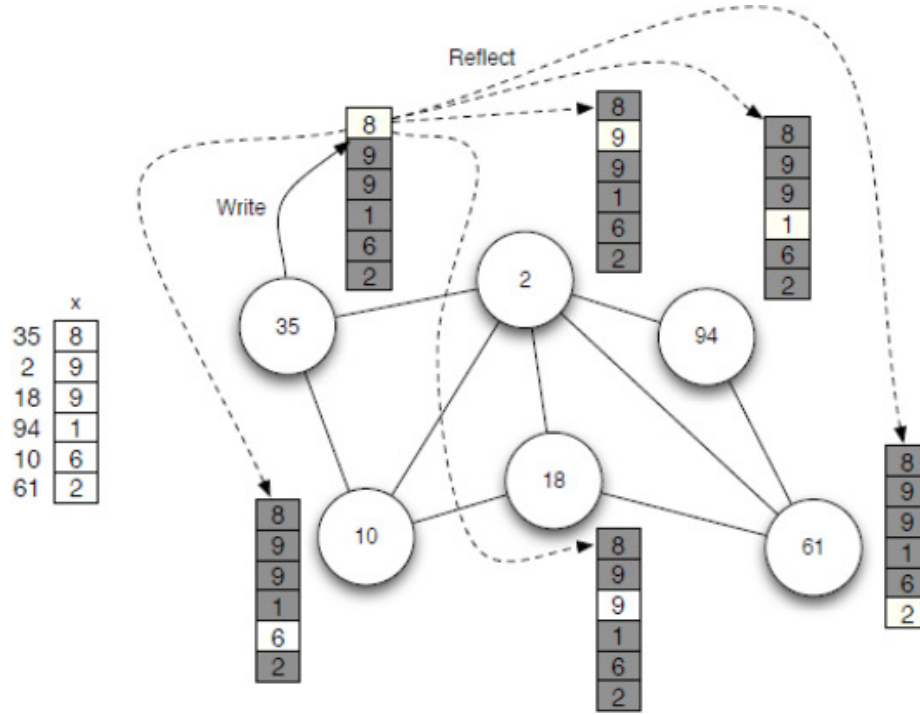


Figure 2.4: Reflected Macrovector: Nodes can read all values in the vector, but can only write to their own value.

memory for parallel systems [32]. It is different from a neighbor-based macrovector because the scope of a reflected macrovector is *globally defined*; it is not different for each node. The reflected representation is beneficial when used by a small *group* of nodes that are relatively close to each other but comparatively far from the base station. In that situation, the cost of sending information to the base station to perform a **max** operation is higher than the cost of transmitting to all other nodes in the group. The reflected representation may also make an operation cheaper when all nodes need the result.

Rule-based Microcode Translation

Given a representation for each macrovector, the decomposer must produce the appropriate microcode for each operation in the macroprogram. For example, the **max** operator must perform different actions when operating over a distributed vector than when operating over a centralized vector. To accomplish this, MacroLab uses a library of microcode templates for each operator and the different representations of its input parameters. Thus, **max** would require three implementations and a binary operator would require 3×3 implementations. This approach of using a library of

operator implementations to deal with different matrix representations has been used before [33] and most of this complexity is hidden from the user.

| | | lhs = M(a₁, ..., a_n) % Synchronous Read |
|--------------|---|---|
| Cnt. or Ref. | R | owner_id = RTS.owner(cur_pc(), M); RTS.notify(cur_pc(), owner_id); |
| | L | node_ids = source_nodes(a1); RTS.wait(cur_pc(), node_ids); lhs = local(M(a1, ..., an)); |
| Distributed | R | if (a1 contains node_id()) then owner_id = RTS.owner(cur_pc(), M); RTS.send(owner_id, local(M(node_id(), a2, ..., an))) RTS.notify(cur_pc(), owner_id); fi; |
| | L | node_ids = source_nodes(a1); RTS.wait(cur_pc(), node_ids); lhs = local(M(a1, ..., an)); |

Table 2.1: For each data representation, the row marked L (for *local*) denotes the code for the mote that will perform the operation (i.e., the locus of synchronization); R (for *remote*) marks the code for all other nodes. **M** is a macrovector; **lhs** and **rhs** are normal vectors. The mote-local representation of **x** is given by **local(x)**. The **owner(PC,M)** function gives the ID of the node requesting the read or write operations on macrovector **M** at location **PC** in the macroprogram.

An implementation of a vector operation will typically consist of multiple functions, each of which is loaded onto the domain of one of the input parameters. Tables 2.1 and 2.2 shows the various implementations of two basic macrovector operations: reading and writing one or more elements in a vector. In this context, the vectors **lhs** and **rhs** are normal vectors and **M** is an n-dimensional macrovector that is being accessed by indices **a1** through **an**. These indices are themselves vectors and index an entire dimension of the matrix **M**. The microcode for these operations is different for each of the three representations discussed in 2.2.1: centralized, reflected, and distributed. For each representation, the operation is divided into two functions: one for the L (*local*) nodes and one piece of code for the R (*remote*) nodes.

For illustrative purposes, the microcode for the synchronous **read** (Table 2.1) and **write** (Table 2.2) operations are shown. The are based on the standard **notify** and **wait** primitives instead of using message passing primitives. **RTS.wait(PC, node_ids)** causes the current node to block until each of the nodes in **node_ids** has called **RTS.notify** with a matching value for **PC**. A call to **RTS.notify(PC, node_ids)** signifies that the caller has “caught up” to a particular program point in the macroprogram. **RTS.notify** blocks until the corresponding **RTS.wait**. This is done

| | | |
|-------------|---|---|
| | | $M(a_1, \dots, a_n) = rhs$ % Synchronous Write |
| Centralized | R | <pre> if (a1 contains node_id()) then owner_id = RTS.owner(cur_pc(), M); RTS.wait(cur_pc(), owner_id); fi; </pre> |
| | L | <pre> node_ids = source_nodes(a1); local(M(a1, ..., an)) = rhs; RTS.notify(cur_pc(), node_ids); </pre> |
| Reflected | R | <pre> if (a1 contains node_id()) then owner_id = owner(cur_pc(), M); RTS.receive(owner_id, local(M(a1, ..., an))); RTS.wait(cur_pc(), owner_id); fi; </pre> |
| | L | <pre> node_ids = source_nodes(a1); local(M(a1, ..., an)) = rhs; foreach (node_id in node_ids) do RTS.send(node_id, local(M(a1, ..., an))); done; RTS.notify(cur_pc(), node_ids); </pre> |
| Distributed | R | <pre> if (a1 contains node_id()) then owner_id = RTS.owner(cur_pc(), M); RTS.receive(owner_id, local(M(node_id(), a2, ..., an))); RTS.wait(cur_pc(), owner_id); fi; </pre> |
| | L | <pre> node_ids = source_nodes(a1); local(M(a1, ..., an)) = rhs; foreach (node_id in node_ids) do RTS.send(node_id, local(M(a1, ..., an))); done; RTS.notify(cur_pc(), node_ids); </pre> |

Table 2.2: For each data representation, the row marked L (for *local*) denotes the code for the mote that will perform the operation (i.e., the locus of synchronization); R (for *remote*) marks the code for all other nodes. M is a macrovector; lhs and rhs are normal vectors. The mote-local representation of x is given by $local(x)$. The $owner(PC, M)$ function gives the ID of the node requesting the read or write operations on macrovector M at location PC in the macroprogram.

to prevent the notifying node from overwriting data before it can be used by the waiting thread. $RTS.owner$ takes the current location in the macroprogram ($cur_pc()$) and returns the node ID of the local node. This ID may be fixed at compile time. Operations over a centralized macrovector are always performed on the node that has the local copy of that macrovector. In other cases, such as neighborhood-based operations, the ID of the local node is unknown until runtime and may change over time.

In principle, this implementation of synchronous `read` and `write` could be used to implement most macrovector operations. Figure 2.3 show implementations of synchronous `max` and `find` functions. The synchronous `max` operation works by reading `A` into a local variable (which is a normal vector), performing the operation, and then writing the results. Thus, this implementation caches the entire macrovector on a centralized node before performing the operation. The code for synchronous `find` assumes the existence of a temporary vector `Temp` created by the translator to store the result of the find in single-column form. Both operations are synchronized because the initial read is synchronized, although this naïve approach incurs a round-trip message between the local node and each remote node.

| | |
|--|---|
| <pre> 1 % smax(A) 2 sread(A, lvar); 3 lmax = max(lvar); 4 write(A, lmax); 5 . </pre> | <pre> 1 % sfind(A(1) = 5) 2 sread(A, lvar); 3 lres = 4 find(lvar(1) = 5); 5 write(Temp, lres); </pre> |
|--|---|

Table 2.3: Local half of naïve implementations of the synchronous `max` and `find` functions.

In practice, many macro-operations will require specialized implementations to further reduce messaging overhead. For example, the `max` operation could be performed using in-network aggregation such as that used in Tiny AGgregation (TAG) [34]. This implementation is a *semantics-preserving* optimization. It is computationally equivalent to the naïve implementation shown above, but would result in lower messaging overhead. The following section discuss optimizations to reduce messaging overhead that are not semantics-preserving.

Reducing Synchronization Overhead

Synchronization is one of the main costs of a MacroLab program; nodes must send messages to indicate that they have “caught up” to a point in the macroprogram, even if they have no useful data to provide. This messaging overhead is necessary to preserve the semantics of the original macroprogram, but many CPS applications do not need strict synchronization for proper operation. Consider the example below, where the intent of this code is to provide a frequently updated maximum light value.

```

1 every(1000) {
2   light = sense(lightSensors);
3   maxLight = max(light);

```

4 }

In this case, synchronizing line 3 is unnecessary since the user is probably not explicitly interested in having `maxLight` represent the maximum for a particular loop iteration. Similarly, the synchronized version of the operation `find(light > 100)` would require a round trip message from all nodes, including those that have `light` values less than 100. An unsynchronized implementation could require messages only from those nodes that have values greater than 100. These optimizations improve parallelism and reduce messaging overhead, but do not preserve the original semantics of the vector operations. To allow users to employ these optimizations, MacroLab must provide both synchronized and unsynchronized versions of each operation. The synchronized versions take function names that start with an *s*: `smax`, `swrite`, `splus`, etc. The usual notations (e.g., `A + B`, `max(A)`) will refer to the *unsynchronized* version of that operation.

It is important to note that not all vector operations require synchronization. Operations over macrovectors generally fall into two categories: *row-parallel* operations and *inter-row* operations. An inter-row operation is any expression that “mixes” macrovector domains. For example, an expression like `max(A)` returns a single-valued result by combining information from all rows of `A`. Inter-row operations require synchronization to be semantics-preserving, but alternative implementations with different semantics can be used in order to reduce overhead. Conversely, a statement like `A = A + B` is *row-parallel*: the operation over any particular row does not require information from any other row, and so the operation can be performed over each row without waiting for `A` and `B` to be fully updated. When multiple row-parallel operations occur consecutively, their execution may overlap.

Generally, inter-row operations are more expensive than row-parallel operations because of the synchronization required. The difference between row-parallel and inter-row operations is evident from the source code, providing the user with a *clear cost model* of the macroprogram and empowering the user to write optimized code that will produce efficient decompositions. Furthermore, the user can control the amount of synchronization overhead by choosing between synchronized and unsynchronized versions of each inter-row operation.

2.2.2 Cost Analyzer

The decomposition process produces many feasible candidate program decompositions. The goal of cost analysis is to predict which candidate will be most efficient for a target deployment. Two

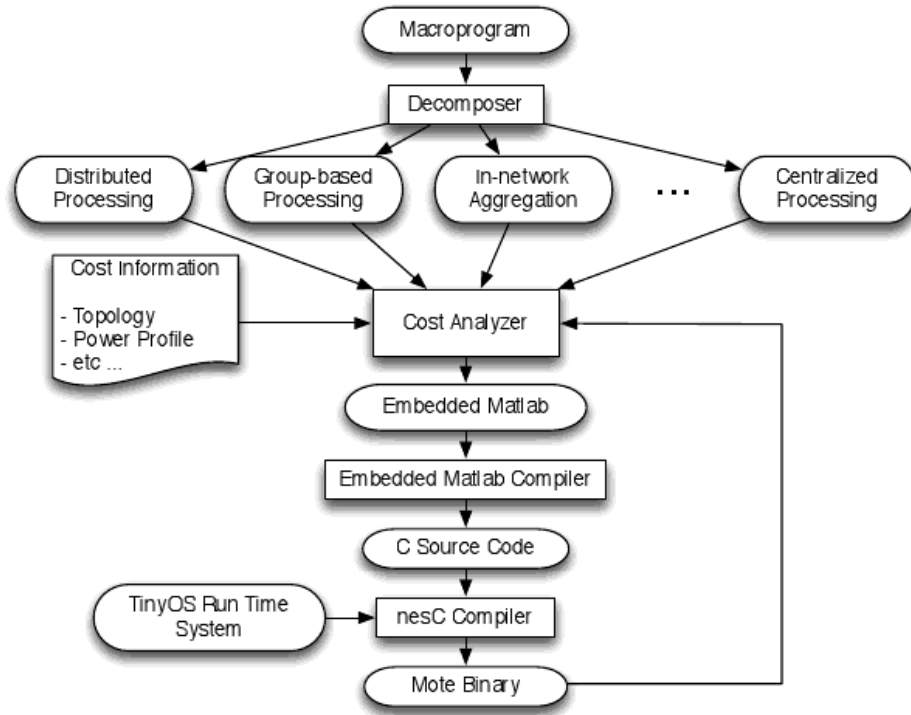


Figure 2.5: MacroLab consists of a decomposer, a cost analyzer, and a run-time system. In this implementation, all possible decompositions of a macroprogram are generated and then analyzed and compare them based on the cost profile of a target deployment.

different techniques for cost analysis are used: compile-time analysis and run-time analysis. As shown in Figure 2.5, an initial decomposition might be chosen using the static analysis until richer run-time profile information is available. Cost analyses can be based on a number of different cost functions such as power, bandwidth, messages, or latency. In this dissertation, only messaging cost is considered.

Static Cost Analysis

The static analysis approximates the true messaging cost of a MacroLab program based on (1) user-provided cost information for messages, (2) sensing event frequencies, (3) a description of the deployment network, and (4) a conservative analysis of the source code to locate network sends. The cost information is provided as a matrix indicating hop counts between nodes. The high-level structure of the static cost analyzer is as follows:

```

1 totalCost = 0;
2 foreach node x in the network do

```

```

3  cost[x] = 0;
4  foreach predicted send in one run of x do
5      cost[x] += hop_count[x,target_of_send] *
               frequency_of_this_event_at_this_node;
6  done;
7  totalCost += cost[x];
8 done;
9 return totalCost;

```

It is assumed that the MacroLab program follows a main event-loop format (as in lines 4–7 of Figure 2.7, lines 6–12 of Figure 2.8, or lines 10–17 of Figure 2.10) and predict the cost for one run through that loop. An intraprocedural dataflow analysis is used to scan the statements in the program’s main loop for predicted sends (line 4).

Each statement that contains a macrovector operation, such as a read from a sensor array, is considered. Based on the decomposition, the operation is analyzed to determine if it involves remote or distributed operations and thus, message sends. If the destination of the message cannot be statically determined, the maximal hop cost from the current node, which is a conservative estimate, is used. The hop cost is weighted by the relative frequency of that event.

Note that it is not possible for a distributed operation to trigger other distributed operations in MacroLab; the user cannot write messaging code directly and all sends and receives are inserted by the decomposer and mediated by the run-time system. There are no message loops to consider and it suffices to consider each node separately to calculate a total predicted cost.

If the user does not have a model of the sensing event frequencies at each node, the analysis assumes events will occur with equal frequency across all nodes for all decompositions. A final cost estimate can be produced that is relative to the number of sensing events. In such a scenario, the cost analyzer does not predict the actual messaging cost of a decomposition but still provides a useful heuristic for distinguishing *between* two candidate decompositions.

Note that a more precise dataflow analysis for predicting statement frequencies (e.g., [35]) could be used. However, the message costs would still have to be modeled, the network topology understood, and the event frequencies predicted. In the experiments presented in this dissertation these were the determining factors and the MacroLab programs themselves were small and easy to analyze.

Run-time Cost Analysis

A run-time analysis to measure the costs of a deployed decomposition can also be performed. The decomposer is used to inject logging code at appropriate locations in the microprogram to count the messages that are actually being sent. Logging code can also be injected to estimate how many messages *would be* sent by other decompositions (as in Tables 2.1 and 2.2). The logged information is periodically analyzed by the cost analyzer to ensure that an efficient version of the implementation is executing. If the currently-executing version is more costly than an alternative, the network can be reprogrammed with the alternative decomposition. Currently, it is not possible to reprogram the network without losing the state of the program that was generated by a different decomposition, but this will be analyzed in future work.

2.2.3 Compilation and Run-Time System

The run-time system supports three operations for

MacroLab microprograms: (1) networking, (2) hardware access, and (3) accessor functions to information such as node ID, current time, location, radio neighbors, etc. The RTS is written as a nesC module and supports the functions shown in Table 2.4. The first three functions are provided directly by the RTS while the last three functions must interface with TinyOS libraries for capabilities such as time synchronization and networking operations. By interfacing with TinyOS, MacroLab leverages an existing suite of distributed algorithms as well as ongoing advances and future software development.

| Operation |
|---|
| getID() <i>returns the node's ID</i> |
| getProperty('property') <i>returns a generic property of a node</i> |
| getNodes('group') <i>returns the current membership in a global group</i> |
| getTime() <i>returns the current global time</i> |
| getNeighbors() <i>returns the current radio neighbors of a node</i> |
| remoteFeval(nodeIDs,funcName,{P1,P2,...,Pn}) <i>remote function invocation</i> |

Table 2.4: The RTS must provide an interface for neighbor discovery, time sync, and remote function calls.

The `remoteFeval` function is a generic messaging interface provided to the MacroLab microprograms. It is similar to Matlab's `feval` function in that it takes a function handle and a set of arguments and invokes that function with those arguments. However, `remoteFeval` invokes the function on a set of remote nodes indicated by the `nodeIDs` parameter. The function name and arguments are marshalled into a packet, sent to those nodes, and unmarshalled before the function is invoked. The `remoteFeval` function can take an arbitrary set of nodeIDs and the RTS decides the best way to send the message. For example, if nodeIDs only contains the ID of the base station node, the RTS sends the message using a standard TinyOS routing protocol. If nodeIDs only contains IDs of neighboring nodes, the RTS sends the message using a local broadcast. In the current implementation, the RTS will flood the message to all nodes for any other set of nodeIDs, but multi-cast algorithms or other routing algorithms could easily be inserted as they are developed.

Access to hardware such as sensors and actuators must be done through new functions provided by user defined hardware drivers. The `BASE_DISPLAY` and `CAMERAFOCUS` functions shown in Figures 2.7 and 2.8 would simply be C functions provided by the user that would set the input/output pins of the microcontroller based on the input parameters. In the case of split-phase functions like `sense`, the driver must declare the name of the callback that will be triggered when the function is complete. The decomposer then generates the appropriate code using this callback to continue execution in the microprogram. The actual light sensor driver used by the code in Figure 2.7 is shown in Figure 2.6.

In order to conserve power, MacroLab enables the TinyOS 2.x low-power listening capabilities [36], which automatically duty cycles the radio and sends messages with long preambles. The RTS dynamically sets the sleep interval of the radio based on the number of messages currently being sent by the application. The entire network starts with a default sleep interval of 100 milliseconds and nodes will flood the network to halve or double the sleep interval when total transmission time is greater than 80 percent or less than 20 percent. Thus, the radio sleep interval for the entire network is dynamically set based on the node with the highest load. This simple algorithm will not always be optimal, but it works well for the existing applications, as shown in Section 2.3.2, more sophisticated adaptive algorithms will be explored in future work. MacroLab programs are executed on Telos [37] nodes, for which the TinyOS libraries automatically use low-power mode when idle.

The MacroLab microprogram, the RTS, and the TinyOS libraries are compiled together into a single binary executable (Figure 2.5) that can run on mote-class devices such as the MICA [38]

```

1 module LightSensorP {
2   provides interface LightSensor;
3   uses interface Read<uint16_t> as Read;
4 }
5 implementation
6 {
7   command void LightSensor.sense() {
8     call Read.read();
9   }
10  event void Read.readDone(error_t err,
11    uint16_t val) {
12    if(err == SUCCESS) {
13      #CALLBACK(val);
14    }
15  }
16 }

```

Figure 2.6: The `LightSensor.sense` function is called by the microcode through the RTS. The decomposer must automatically replace `#CALLBACK` with an appropriate function to continue micro-program execution.

and the Telos. The microprogram generated by the decomposer is written in Embedded Matlab, a simplified form of the Matlab syntax that does not support dynamic typing or dynamic memory allocation. This is compiled down to C code by the Embedded Matlab compiler, provided by The MathWorks [39]. This C code is then compiled together with the nesC RTS module and the TinyOS libraries by the nesC compiler.

2.3 Evaluation

MacroLab is evaluated in four parts. First, the programming abstraction is evaluated by showing that it is expressive enough to implement canonical CPS applications, such as data collection and object tracking. Second, the overhead of running MacroLab programs is compared with similar programs written using nesC and TinyOS. Third, a MacroLab application is deployed in multiple different scenarios and measure the effect of DSCD on message cost. Finally, the accuracy of the static cost analyzer is evaluated.

2.3.1 Expressiveness of the Abstraction

The expressive power of the programming abstraction is evaluated by showing that it can be used concisely to implement two canonical CPS applications: *tree-based data collection* in Surge [1] and

object tracking in the Pursuer-Evader Game (PEG) [40]. These two applications were selected because they represent basic algorithms that have been incorporated into many other CPS applications. Table 2.5 presents a comparison of the number of lines of code necessary to implement Surge and PEG in MacroLab as compared to their original implementations in nesC/TinyOS. The number of lines of code for the nesC/TinyOS implementations of Surge and PEG were cited from previous publications [19, 41]. The MacroLab applications are about one-fiftieth the size of equivalent nesC/TinyOS implementation in terms of lines of code. This ratio is typical of the difference in size between equivalent Matlab and C programs. MacroLab builds a considerable amount of logic such as routing algorithms and vector operations into the run-time system. MacroLab dramatically reduces the amount of code and therefore the amount of development and maintenance time required to implement basic CPS applications.

The MacroLab programming abstraction is suitable for many application domains, but it cannot express algorithms that require explicit message passing. For example, it cannot easily be used to implement routing protocols, time synchronization protocols, or other distributed middleware libraries. Instead, all of these operations must be incorporated into the run-time system. Thus, MacroLab programs are limited to distributed operations that can be neatly stored in a library and provided by some interface. This is not a restricting requirement for most CPS software. However, it can make it difficult to provide application-specific distributed operations. Developers must encapsulate such functionality by extending the run-time system. Code movement in systems like Agilla [42] and EnviroSuite [43] might be difficult to implement in MacroLab.

| | MacroLab | nesC/TinyOS |
|-------|----------|-------------|
| Surge | 7 | 400 |
| PEG | 12 | 780 |

Table 2.5: Comparison of lines of code required for equivalent functionality in two basic CPS applications.

Surge

Surge is a simple application that periodically collects sensor readings from all nodes and routes them back to a base station. The data collection aspect of Surge has been widely utilized in many sensor network applications. In fact, many sensor network applications, such as [44–46], involve primarily data collection and in many other applications, such as AlarmNet [47], data collection

plays an important role. Thus, it is essential that an abstraction for sensor network programming make it easy to implement data collection applications. Figure 2.7 shows the source code for Surge written in MacroLab. In line 1, the run-time system is instantiated and in lines 2 and 3, it is used to instantiate a vector of light sensors (one for each node in the network) and a macrovector to hold the light values. Line 4 is the beginning of a loop that occurs every 1000 milliseconds. The light sensors are read in line 5 and the values are displayed at the base station in line 6.

```

1 RTS = RunTimeSystem();
2 lSensors = SensorVector('lightSensor','uint16');
3 lightValues = Macrovector('uint16');
4 every(1000)
5   lightValues = sense(lSensors);
6   BASE_DISPLAY(lightValues);
7 end

```

Figure 2.7: Surge reads sensor values and displays them at the base station. `BASE_DISPLAY` is implemented within the RTS and sends a message to a base station for display.

There is hidden complexity in the interaction between lines 5 and 6. The decomposer identifies that the sensor resources are on the nodes while the `BASE_DISPLAY` function is only available on the base station. It therefore infers that the information created in line 5 must be routed across machine boundaries in order to be used in line 6 and the compiled version of the code automatically invokes the routing algorithm via the `remoteEval` interface described in Section 2.2.3. The high-level algorithm can be expressed in seven lines of MacroLab. The automatically generated microcode that runs on the nodes is closer in size to the nesC/TinyOS implementation.

Pursuer-Evader Game (PEG)

The Pursuer-Evader Game (PEG) is a distributed CPS application that detects and reports the position of a moving object within a sensor field. It is a canonical application in the field of robotics research where multiple robots (the pursuers) collectively determine the locations of one or more evaders and attempt to corral them. This application has been adapted by the sensor network community [40,48,49] and represents the algorithmic capabilities necessary for a number of canonical applications such as object tracking and in-network filtering of false positives. It is widely used as an application for the evaluation of sensor network programming abstractions and architectures [12,19,22] since it is one of the largest TinyOS applications and utilizes a number of relatively sophisticated concepts for sensor networks. Figure 2.8 shows an implementation of PEG in MacroLab in which the

```

1 motes = RTS.getMotes('type', 'tmote')
2 magSensors = SensorVector(motes, 'magnetometer');
3 magVals = Macrovector(motes);
4 neighborMag = neighborReflection(motes, magVals);
5 THRESH = uint8(500);
6 every(1000)
7   magVals = magSensors.sense();
8   active = find(sum(neighborMag > THRESH, 2) > 3);
9   maxNeighbor = max(neighborMag, 2);
10  leaders = find(maxNeighbor(active) == magVal(active));
11  CAMERAFOCUS(leaders);
12 end

```

Figure 2.8: Every 1000 ms, nodes take a reading from their magnetometers and share the values with their neighbors. If more than three nodes in a neighborhood sense a magnetometer value about a threshold, a leader is elected from among them and a camera is focused on it. `CAMERAFOCUS` is implemented within the RTS and sends a message to the camera, which is at the base station.

network routes the location of the evader to a camera which visually follows the evader. In lines 1–3 the code initializes the `motes` vector of node IDs, the `magSensors` vector of magnetometer sensors, and the `magVals` macrovector of magnetometer readings. In line 4, a $n \times n$ `neighborReflection` macrovector is created where each element i, j has a cached copy of the j th element of `magVals` if i is a neighbor of j , and an invalid value otherwise. In the main loop, every 1000 ms, line 7 reads from all magnetometer values and line 8 creates an `active` vector with the IDs of all nodes that have at least three neighbors with values higher than `THRESH`. Since `sum` returns the sum over columns by default, the second parameter, 2, to `sum` indicates the sum across rows to be calculated since each row contains a particular node’s neighborhood. Line 9 creates a `maxNeighbor` vector with the highest sensor value in each node’s neighborhood. Here too the 2 as the second parameter indicates that the maximum value across nodes should be calculated instead of across columns. Line 10 creates a `leaders` vector, which contains the IDs of those nodes that have the highest sensor value in an active neighborhood. Finally, line 11 uses a proprietary function called `CAMERAFOCUS` to focus all available cameras on the leader nodes, using a pre-defined mapping from nodeID to location. The user, or camera manufacturer, would need to write the `CAMERAFOCUS` function, which is basically a hardware driver, while the standard Matlab functions such as `find`, `sum`, and `max` are provided by the MacroLab system. The hardware drivers for the sensors are also provided by the MacroLab system by using the TinyOS operating system. The purpose of this example is to demonstrate that MacroLab can concisely represent efficient, neighborhood-based, in-network processing.

2.3.2 Performance Overhead

To evaluate the performance overhead of MacroLab, the resource consumption of the two applications described in Section 2.3.1 is measured and compared against existing applications written in nesC/TinyOS. MacroLab programs are very similar to TinyOS programs in terms of memory footprint, execution speed, and power consumption.

The program size and heap size of two MacroLab programs is shown in Table 2.6, along with those of several existing TinyOS programs. The MacroLab programs actually have a smaller memory footprint than their corresponding TinyOS implementations, both in terms of program memory and RAM. While the TinyOS versions may be somewhat more feature rich, the MacroLab versions are still not much larger than extremely simple TinyOS applications such as **SenseToRfm**. Table 2.7 shows the breakdown of the MacroLab programs' memory footprints. This information was collected by examining the symbol table of the final binary executables. Therefore any variables or code removed by compiler optimizations are not included. The vast majority of the program size of MacroLab applications (approximately 17KB of ROM and 600 bytes of RAM) is due to imported TinyOS libraries for multi-hop routing, access to the Analog-to-Digital Converter (ADC), and the Timer. The run-time system module requires 558 bytes of program memory and 66 bytes of RAM. For both PEG and Surge, the application logic itself required less than 1.3KB of ROM and 150 bytes of RAM.

| Application | Program Size | Heap Size |
|-----------------------|--------------|-----------|
| TelosB | 49,152 | 10,240 |
| MICAz | 131,072 | 4,096 |
| Blink | 2,472 | 38 |
| CountRadio | 11,266 | 351 |
| Oscilloscope | 9,034 | 335 |
| OscilloscopeRF | 14,536 | 449 |
| SenseToRfm | 14,248 | 403 |
| TOSBase | 10,328 | 1,827 |
| MacroLab_Surge | 19,374 | 669 |
| SurgeTelos | 24,790 | 911 |
| MacroLab_PEG | 18,536 | 770 |
| PEG [40] | 61,440 | 3,072 |

Table 2.6: Program and heap size comparison for common TinyOS applications and two MacroLab applications for TelosB nodes.

The maximum run-time stack sizes of the MacroLab and nesC Surge implementations are compared. Memory was filled with a special reference pattern, and a program run for 600 seconds.

Then the high water mark for stack growth was computed by looking for the last byte not containing the reference pattern. MacroLab’s RTS layer introduces additional functions and could therefore potentially require more memory for the stack. However, aggressive function inlining by the nesC compiler causes the function call depth to be almost the same for both applications. The TinyOS version requires 120 bytes while the MacroLab version requires 124 bytes for the stack, as shown in the last column of Table 2.8.

| Application | TinyOS | RTS | MacroLab |
|----------------|-----------|--------|----------|
| MacroLab_PEG | 16714/579 | 558/66 | 1264/125 |
| MacroLab_Surge | 15714/579 | 558/66 | 1144/24 |

Table 2.7: A breakdown of the amount of flash/RAM in the TinyOS libraries, RTS, and program logic of MacroLab applications

To compare the execution speed of a MacroLab program with a TinyOS program, the time to execute Surge from the beginning of the loop when the node reads the sensor until the radio has accepted the message for transmission is measured. These times do not include the Media Access Control (MAC) and transmission delays. This time was measured for both the MacroLab and TinyOS implementations using an oscilloscope. The first two columns of Table 2.8 show that the total execution time was about 18 milliseconds for both programs, but the MacroLab program takes about 3 percent longer (0.5 milliseconds). The CPU was idle for most of the execution, waiting for the ADC to return a value. The non-idle time for the MacroLab program was 705 microseconds compared with 361 microseconds for the TinyOS program. Thus, the MacroLab program requires almost twice as many instructions to be executed as the TinyOS program. This is a large multiple, but the effect on overall execution time and power consumption is small.

The power consumption of both implementations of the Surge application is shown in Figure 2.9. In both cases, power consumption was measured using an oscilloscope on a node that was forwarding messages from exactly two children. The period with which Surge sampled the sensor and forwarded the message to the base station was varied from 100 milliseconds to 10 seconds. The results show that the average power consumption over a sample run of 100 seconds is nearly identical for both implementations, even as the sampling frequency changes by over two orders of magnitude. This evidence suggests that MacroLab programs can match TinyOS programs in terms of power efficiency.

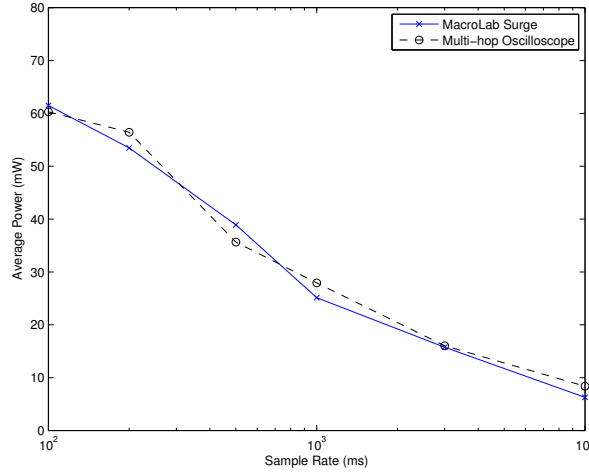


Figure 2.9: Oscilloscope power measurements of MacroLab and nesC Surge implementations.

| Application | Execution | CPU | Stack |
|-----------------------|-----------|---------|----------|
| Surge | 17.7msec | 361usec | 120bytes |
| MacroLab_Surge | 18.2msec | 705usec | 124bytes |

Table 2.8: An evaluation of the execution time of the application, logic (CPU), and maximum consumed stack memory.

2.3.3 Effect of DSCD on Performance

To evaluate the effect of DSCD on the performance of an application in multiple scenarios, a *bus tracking* application (Figure 2.10) was implemented. In order to easily modify the deployment scenario, this part of the evaluation was performed in simulation. In the bus tracking application, each bus stop records arrival times for the buses and computes estimated arrival times for all other buses. The application logic is shown in Figure 2.10, which maintains state about the last time a bus was seen at every stop, the time it takes to travel from each stop to every other stop, and the estimated time that each bus will next arrive at every stop.

First, a bus is sensed at each stop and collect a time stamp of the bus arrival as `busTime`. The *arrivals* matrix stores the last time before now that each bus arrived at each stop and *travelTime* is updated to be *busTime* minus *arrivals*. In other words, the travel time between stops is estimated to be the current time that each bus arrived less the last time that bus was seen at every other stop. *Arrivals* is then updated with the current arrival time of the bus. Next, the *estimates* vector is updated to be *travelTime* plus *busTime*. The predicted arrival time for each bus is estimated to

```

1   RTS = RunTimeSystem();
2   busstops = RTS.getNodes('stopnode');
3   buses = RTS.getNodes('bus');
4   estimates = Macrovector(busstops, length(buses) , 'uint16');
5   arrivals = Macrovector(busstops, length(buses) , 'uint16');
6   travelTime = Macrovector(busstops, length(busstops), length(buses) , 'uint16'
   ');
7   busSensors = SensorVector('BusSensor',busstops, 'uint16');
8   routes = uint8([1 2 3 4], [ 5 6 7 8]); %Example routes
9
10  while(1)
11      [busID,r] = sense(busSensors);
12      busTime = RTS.getTime();
13      travelTime(routes{r},routes{r},busID)[1,3] = busTime - arrivals(routes{r}
   }, busID);
14      arrivals(routes{r},busID)[1,2] = busTime;
15      estimates(routes{r},busID) = travelTime(routes{r},routes{r},busID)[2,3] +
   busTime;
16      BASE_DISPLAY(estimates(routes{r},:));
17  end

```

Figure 2.10: MacroLab code for the bus tracking application.

be its travel time plus the last time it was seen at all stops. Finally, the estimated arrival times are displayed to potential passengers using the `BASE_DISPLAY` operation. The matrix operations in this application make heavy use of the dot product notation described in Section 2.2.1 for conciseness and efficiency.

MacroLab’s row-level parallelism allows the matrix operations to occur in parallel. In this particular application, the program runs correctly without using the synchronized implementation of any inter-row operations, and so the assignment in line 15 does not block until all values are collected. Each row can be processed in parallel as buses arrive at different stops.

This application is evaluated in four scenarios: 1) the estimated bus arrival times are displayed to the passengers on a website which is updated from a centralized base station, 2) the estimated bus arrival times are displayed to the passengers at each bus stop, 3) the base station radio range is increased substantially for better coverage, and 4) an additional bus route is added by the bus company. The results show no decomposition is best for all target deployments and choosing the correct decomposition can reduce messaging costs by an average of 44 percent.

MacroLab can optimize for a number of cost metrics (such as latency, power consumption, or message cost) by using a cost profile and a cost analyzer that analyzes the cost of each program decomposition for a target deployment scenario. In this evaluation, only the total number of messages

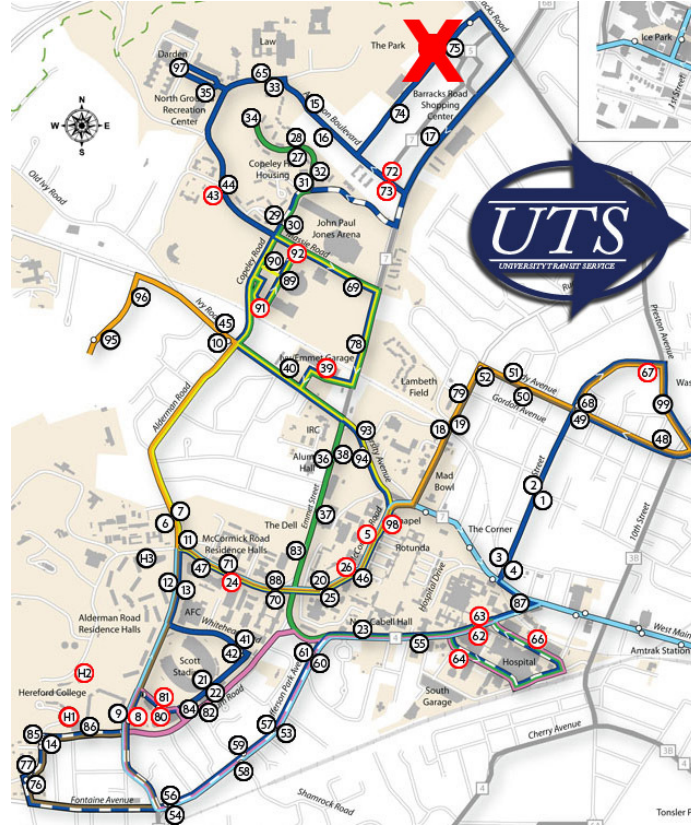


Figure 2.11: UTS Bus Routes at U.Va. The base station is indicated as a cross at the top of the figure.

that must be sent by the network to achieve the global objective is optimized.

Experimental Setup

The test scenario consists of actual bus routes at the University of Virginia, provided by the University Transit Service (UTS), shown in Figure 2.11. It is assumed that each bus stop has a mote and can sense the bus currently at its stop. The cost profile of the test scenario was created based on the actual locations of the bus stops and assuming a reliable communication range of 500 meters

In order to test this wide variety of deployment scenarios, a Matlab-based simulator was built for this part of the evaluation. The simulation environment for MacroLab only requires a slight modification to the RTS in order to support function calls into the simulator instead of directly to the nodes. The simulator runs code similar to microcode that would run on a mote-based RTS. In this experiment, only two decompositions are evaluated: a centralized and distributed decomposition.

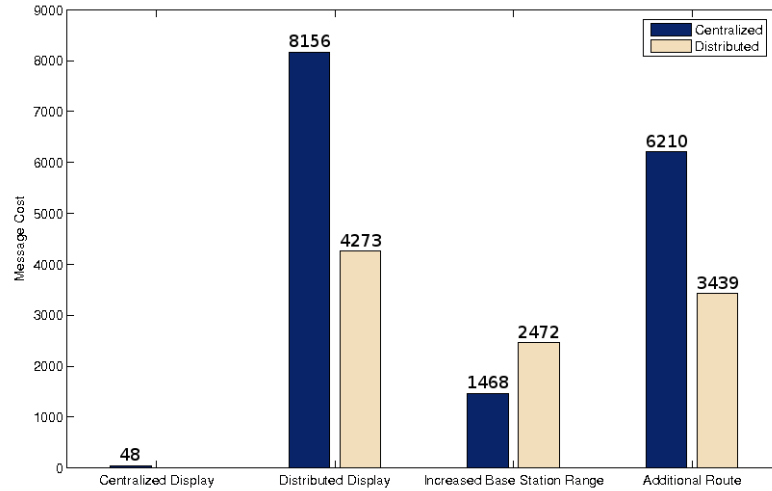


Figure 2.12: Neither decomposition is best for all deployment scenarios. Small changes in the deployment scenario changes the optimal implementation between centralized and distributed.

Communication between nodes is provided by the RTS. Based on the cost profile of the network, the simulation observes the total message cost for each decomposition and scenario. The cost matrix is computed from the topology of the scenario and encodes the cost to send messages between pairs of nodes. The RTS currently supports point-to-point routing but more routing algorithms can be added as they are developed.

Scenario 1: Website Display

In the first scenario, the bus information is recorded and displayed on a website using a centralized base station. This is accomplished by using the `BASE_DISPLAY` operation which can only be executed in a centralized fashion. In order to accomplish this version of the application, the nodes will sense buses and send their data directly back to the base station which will maintain state in all of the macrovectors and perform all of the vector computations. The messaging cost of this application is the same as the Surge application. Only one bar is shown in Figure 2.12 because the `BASE_DISPLAY` operator is only defined for centralized decompositions. By running this decomposition in simulation, the cost of running the application using this scenario is found to be 48 messages over a 30 minute simulation.

Scenario 2: Bus Stop Displays

Changing `BASE_DISPLAY` to `MOTE_DISPLAY` allows the application to show the data at each bus stop rather than at the base station. Sending and receiving bus arrival information between all nodes within each route totals 8156 messages for the centralized implementation and 4734 for the distributed version; the distributed decomposition is almost twice as efficient. The large increase in the number of messages is due to the fact that each time a bus arrives at a stop, its arrival time must be transmitted to all the stops in the route. In the centralized implementation, this requires $O(n^2)$ messages per route, where n is the number of stops on that route. This cost increases dramatically for routes that are far from the base station. All nodes must send each bus arrival event all the way to the base station which must then forward the message back to each node on the route individually.

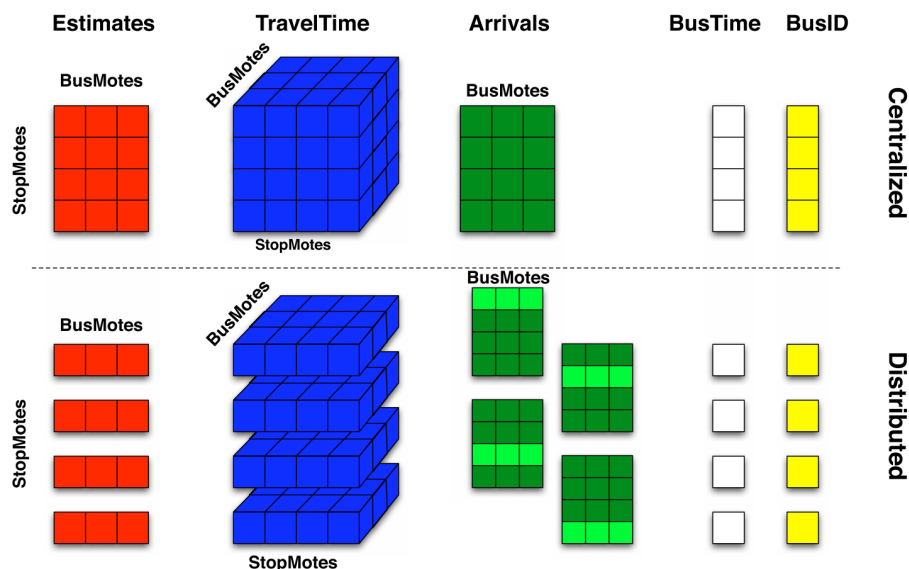


Figure 2.13: Macrovectors in centralized and distributed implementations.

The cost of the distributed implementation does not incur this overhead. Each node forwards bus arrival events to all other nodes on its route; it does not need to transmit back and forth to the base station. If a stop is on two or more routes, it forwards the message to all other stops on all such routes. In the compiled code, this is achieved by distributing all vectors in the program and making the `arrivals` vector a reflected vector across all nodes on a route. The difference in macrovector storage between the centralized and the distributed decompositions is shown in Figure 2.13. For this target deployment scenario, the messaging cost of the distributed decomposition is about 50 percent

lower than the centralized decomposition.

The only change to the bus tracking application in this scenario versus Scenario 1 is that the `BASE_DISPLAY` function was changed to `MOTE_DISPLAY`, changing the library function from a centralized operator to a distributed operator. It should be noted that if this were a TinyOS application, a completely new version of the program would have to be implemented in order to make this change. Thus, small changes in the program can result in large changes to the cost profile of each decomposition. In MacroLab, the single line addition is handled by the decomposer and the RTS in order to choose the optimal solution.

Scenario 3: Increased Base Station Range

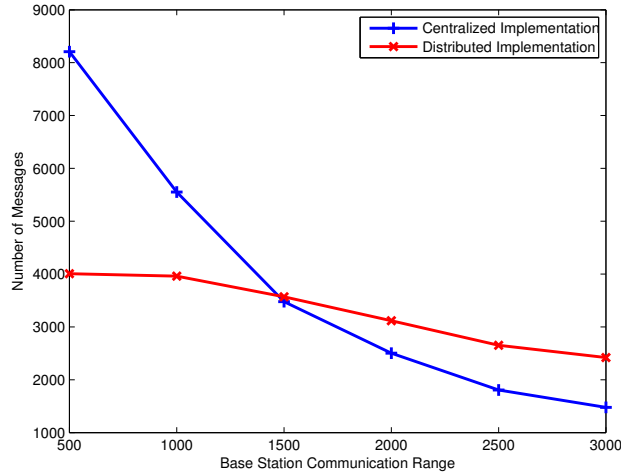


Figure 2.14: Changing the base station range changes the balance between the two decompositions.

In the third deployment scenario, a high-gain antenna is added to the base-station which increases the coverage of the base station and changes the cost profile of the network. Figure 2.12 shows that adding the antenna reduces the cost of messages dramatically. This cost reduction is because the increased range allows all of the nodes to more cheaply communicate with the base station. However, this change affects the centralized decomposition to a greater extent. This is because all nodes in the network use the base station link in the centralized implementation while in the distributed implementation, only the nodes that can opportunistically route through the base station to reduce messaging costs to other nodes on the route will actually do so. Increasing the base station range increases the number of nodes that opportunistically route through the base station, reducing

messaging costs but not as dramatically as in a centralized decomposition. Figure 2.14 shows that for more than a 1500 meter range, the centralized implementation gives superior performance.

In this scenario, a centralized decomposition costs 1468 messages while the distributed version costs 2472 messages. This is a reversal of the cost trade-off in the previous scenario. Small changes to radio hardware can lead to large changes in the cost profile of the target deployment. Redesigning a TinyOS program to be efficient given this new cost profile would be difficult, but the MacroLab framework makes this change automatically.

Scenario 4: Additional Route

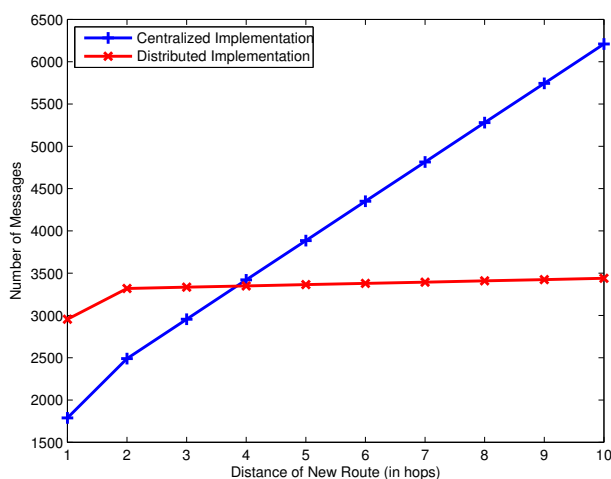


Figure 2.15: Adding a new route at various distances changes the balance between the two decompositions.

In the fourth deployment scenario a route that runs from the main campus to a new location 2500 meters away from the base station is added. Figure 2.15 shows the distributed and centralized costs as a function of the number of hops from the base station. As the number of hops increases, the centralized decomposition must send messages farther to reach the base station, while the cost of the distributed decomposition does not change. Once this messaging cost to and from the base station exceeds the cost of sending messages directly to the other nodes in the route, it is more expensive to utilize the centralized implementation.

Figure 2.12 shows that at an additional 4000 meters, the distributed version becomes more efficient than the centralized decomposition. The centralized cost is 6210 and the distributed cost

is 3439. Because the new route is farther away from the base station, communication with the base station becomes more expensive. This scenario demonstrates that small changes to the network topology can cause large changes to the cost profile of the target deployment.

2.3.4 Accuracy of Static Cost Analyzer

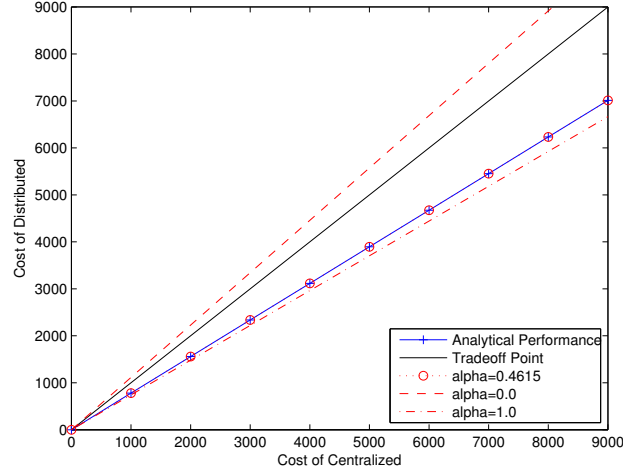


Figure 2.16: Estimated and measured messaging costs. The parameter α is the ratio of buses on long routes versus those on all other routes. The actual distribution of bus detection events can change the true messaging cost within a narrow range around the statically estimated cost.

The static cost analysis (Section 2.2.2) can make use of information about expected event frequency. For simple event-loop MacroLab programs, the presence and quality of this event frequency information is the determining factor in the accuracy of the analysis. In the worst case, such information is not available. For the bus tracking application, lacking any other information, the static analysis assumes that each bus stop observes a bus event with the same frequency. However, this is not necessarily true and there will be some discrepancy between the cost estimated by the static analysis and the actual cost of the decompositions. For example, if a larger percentage of buses appear on very long routes, this will increase the cost of the distributed decomposition relative to the static estimate. On the other hand, if many buses appear on routes far from the base station, this would increase the cost of the centralized decomposition.

Figure 2.16 shows static estimated costs as well as dynamic measured costs as the distribution of buses and therefore sensed events within the network changes. To highlight the differences between

the centralized and distributed implementations, the evaluation focuses on the longest routes. The parameter α is defined as the ratio of buses on the long routes versus those on all other routes. The crossover point for centralized and distributed message costs is represented by the $y = x$ line. Ratios above the crossover line will be optimally run as centralized applications, those below as distributed. By computing the ratios for various scenarios, the effect of the change on the total cost of the application, and whether it will cause a switch in the optimal implementation, can be seen. The bounds of the application are plotted by the $\alpha = 0$ and $\alpha = 1$ lines, in which either all buses or no buses appear on the longest routes. The static cost analysis algorithm is quite accurate for this application and is not extremely sensitive to the assumptions made about the frequency of events; it will only be incorrect with extremely non-uniform event distributions. In such cases, the run-time system can dynamically monitor changes or errors in the cost profile of the target deployment. As shown in Figure 2.5, such dynamic cost information can be fed back into the cost analyzer which can decide to reprogram the network.

2.4 Conclusions

MacroLab’s approach of *deployment-specific code decomposition* addresses a central question in the area of Cyber-Physical Systems software design: should programs be implemented in a centralized or distributed fashion? Most early sensor network research focused on “*localized* algorithms – where sensors only interact with other sensors in a restricted vicinity, but nevertheless collectively achieve a desired global objective” [50]. For example, early object tracking applications argue that neighborhood communication and local processing are necessary to efficiently filter false positives [12] and services like TAG use *in-network aggregation* to calculate network statistics *en route* to decrease message passing [4]. More recently, several architectures have proposed the use of centralized algorithms to control distributed systems. *Marionette* allows the user to write a centralized Python script to control all nodes in a network [18]. It argues that centralized algorithms are easier to write and debug and that, once debugged, functionality can be *migrated* to the sensor nodes for efficiency reasons if necessary [7]. The *Tenet* [22] architecture takes a stronger stance by arguing that all application-specific code should *always* execute on master devices while sensor nodes should be restricted to a small set of predefined operations on locally-generated data. The rationale here is to separate the application code from the networking code so that changes in the application do

not cause cascading changes to the networking middleware.

MacroLab demonstrates that programs can actually be implemented in both a centralized *and* a distributed fashion. The architectural question is reframed from *where code should execute* to *how code should be written*. The central tenet of MacroLab’s architecture is that all application-specific logic should be contained in a macroprogram and all distributed operations must be contained as libraries in the run-time system. When code is written in this way: (1) the decomposer and the run-time system can choose the manner of implementation that provides the best performance in terms of cost metrics like bandwidth, power, and latency, and (2) the user is free to write *deployment-independent* programs that are simple, robust, and easy to understand. In future work, new macrovector representations will be used to decompose programs into many points on the spectrum between purely centralized or purely distributed code, such as hierarchical or group-based data processing, and in-network aggregation.

Chapter 3

MDB: The MacroLab Debugger

MacroLab is a macroprogramming abstraction aimed at simplifying application development for CPSs, yet programming is only a fraction of the complete application development life cycle. Many studies have shown of the time spent developing traditional computer programs, 50% is typically spent on testing and debugging [51]. This percentage is expected to be much larger for CPSs due to the inherent complexity of such systems. While many macroprogramming systems have been presented to ease application development for CPSs, no solution for debugging such systems have been presented. The MacroLab Debugger (MDB) is presented as one approach to simplify CPS application debugging.

MDB is the first debugger that allows the user to navigate and view the distributed state of a program in terms of high-level macroprogramming abstractions. The goal of MDB is to provide a source-level debugging interface for macroprograms with which the user can place breakpoints, step through the code, and inspect the values of variables, thus making macroprogramming a feasible approach to CPS application development by providing a more complete tool chain. MDB allows the user to debug a single macroprogram, navigating execution and viewing system state in terms of high-level operations and data structures. This alleviates the need to debug the node-level programs running on each node, including details such as radio messages and hardware interrupts. The key challenge for MDB is that nodes can execute *asynchronously*: each node may be executing a different part of the macroprogram at any given time, and nodes may execute the parts of any given distributed operation at different times. For this reason, MDB provides two different types of views: *logically synchronous views* depict distributed state where all nodes are executing the same line of

code, and *temporally synchronous views* depict the distributed state at a given time.

It is demonstrated that debugging macroprograms is both easier and more efficient than debugging node-level programs. Macrodebugging is easier than low-level debugging for the same reason that macroprogramming is easier: high-level abstractions. Also, macrodebugging is more efficient than low-level debugging: less information needs to be collected about system execution to provide visibility into execution. MDB is a *post-mortem* debugger, which means that it collects data about the system at run time and allows the user to inspect program execution after the logs are retrieved. While MDB is implemented and evaluated using MacroLab, the underlying principles of MDB can be applied to other macroprogramming systems, and at least some of them can be applied to on-line debugging, as discussed in Section 3.5.

3.1 Background and Related Work

A plethora of debuggers and debugging abstractions have been created for distributed computing. For example, source-level debuggers allow the user to halt and step through execution on each individual machine [52–54]. Distributed breakpoints [55–57] and snapshots [58] allow the user to stop all nodes in a particular consistent state across the network. Some debuggers can be programmed so that they interact with and analyze the program as it executes without user interaction [59–61]. Others provide logical or SQL-like query languages for searching, inspecting, and analyzing state or execution flow [62–64]. Each of these systems provides a high-level abstraction for debugging, but are not designed to work with systems that provide a high-level abstraction for programming; these debugging systems allow the user to inspect node-local actions and messaging protocols which are abstracted away by macroprogramming systems.

Many debugging systems have been implemented in ways that accommodate the strict resource requirements of CPSs. For example, Marionette [18] and Clairvoyant [65] provide access to source-level symbols, while Declarative Tracepoints [66] and Wringer [67] provide a programmable interface for describing debugging operations that can be downloaded and executed on the nodes at run-time. DustMiner [68] and LiveNet [69] eavesdrop on messages in the network for visibility into network operations without consuming node resources. EnviroLog [70] logs some non-deterministic events to produce efficient in-network execution replay. Sympathy collects a small amount of data to identify the cause of network failures [71]. However, all of these systems require the inspection of node-local

actions or message traces. The primary contribution of MDB is to avoid this by debugging a CPS using high-level distributed abstractions provided by macroprogramming systems.

MDB is a *static* debugger; it collects execution traces and the debugging process is off-line or *post-mortem*. This approach has long been used for distributed systems because on-line debugging can change the timing characteristics of execution, making it difficult to analyze message races. Most off-line distributed debuggers use *execution replay* to recreate the system state at a specific point in the original execution, in which the debugging runtime records only high-level events and re-executes (or simulates) the intermittent code where necessary to recreate the full system state [72,73]. Execution replay incurs a debug-time cost of recreating the state, and some systems must use checkpointing to limit the amount of replay required [72].

In contrast, MDB avoids most of the cost of replay by logging all macrovector writes, creating a data trace rather than an event trace. This approach has been known to be possible in traditional distributed systems, but not practical. Mall notes that data traces are impractically expensive but shows that the cost can be reduced to some extent with a technique called *inverse statement analysis* [74]. Maruyama et al. state that *data replay* is still uncommon due to its high cost but shows that it is becoming possible with the increasing capacity and decreasing cost of storage [75]. MDB is unique in that it uses data traces because they are more efficient than event traces for its application domain, as shown in Section 3.6.2.

Several debugging abstractions have been created for use during execution replay. For example, TraceView [76] allows the user to visualize event traces. Garg et al. detect weak unstable predicates using traces [57]. Kilgore, et al. replay and change the order of messages to detect *race messages*, where the order in which messages are received change the results of the distributed computation [77]. Finally, D3 [78] allows the user to write a query in a high-level declarative language called NDLog to create a data model of the logs. D3 then applies the query to the incoming data that conforms to the model. Many of these abstractions are similar to MDB’s ability to visualize data, find logical conditions in the network, reorder messages, or analyze the history of variables. One key difference is that MDB’s abstractions were specifically designed for use with data traces, whereas the abstractions above were designed for use with event traces. Another difference is that MDB provides these abstractions on high-level, abstract distributed data structures that are specified in a macroprogram, while the existing systems are applied using symbols from the programs of each individual node.

Debugging on traces allows MDB to provide time travel capabilities for users. This allows the user to go to step back through states, or time, in order to identify the root cause of a bug. A number of time travel debuggers have been proposed for traditional computer programs. Bhansali et al. allows Windows applications to be debugged by running them backwards in order to figure out the root cause of the problem [79]. The Omniscient Debugger (ODB) records every change in every accessible object or local variable in each thread of a Java program to allow a user to navigate backwards in time and look at objects, variables, and method calls [80]. ReVirt executes an application in a virtual machine which logs below the virtual machine so that sufficient information can be logged to replay and analyze bugs [81]. These techniques are not feasible in the domain of CPSs due to its distributed resource constrained nature.

There have been a number of advances in debugging sequential programs. One such advance is delta debugging [82] which automatically minimizes test cases by comparing code differences between two versions of a program and tries to locate changes which cause errors by replacing code from a failed version with code from a succeeding version. Chianti [83] explores many version of a program through a set of tests. Model-based debugging [84], which attempts to automatically locate defects in a program by modeling the program from its source code is another area that has seen considerable research recently. Most of these techniques attempt to automate the task of failure localization. While such an approach is possible for certain classes of failures for CPSs, such as communication failures in data collection applications which Sympathy attempts to localize, the distributed, event-driven nature of CPS applications make them too complicated for such a technique to be utilized to identify failures in general.

3.2 Three Types of Macroprogramming Bugs

This section describes several possible bugs that could appear in a macroprogram using the application presented in Figure 2.8. Although the program is very simple, several bugs are possible due to factors such as human error, message loss, and message races. This section describes three bugs that are representative of the types of bugs that might appear in a typical MacroLab program. These bugs will be used in Section 3.3 to motivate the design of the MDB interface.

Bug 1 – Logical Error: A logical error occurs when the logic of the program is specified incorrectly, perhaps due to a typographical error or a poor understanding of the application by the human user.

For example, on line 5 of the code in Figure 2.8, the user could accidentally set `THRESH` to be 5000 instead of 500, or the user could mistakenly write line 9 as:

```
>> active = find(sum(magVals > THRESH, 2) > 3)
```

Either one of these typos would result in the `active` variable always being an empty vector, in which case no node would ever be elected a leader in the network. Either of these bugs would produce the same manifestation: that moving objects are never detected by the application.

Bug 2 – Configuration Error: A configuration error occurs when the logic of the application is correct, but does not match the details of the deployment topology or physical stimuli. For example, line 6 in the example macroprogram reads from the sensors every 1000ms, but the objects that are being tracked may move past the nodes much more quickly than that. Similarly, the example program requires at least three neighboring nodes to detect a mobile object, but the nodes may be spaced out so far from each other that only one or two nodes ever detect the mobile objects. Either of these bugs would produce the same manifestation: that moving objects are sporadically or inconsistently detected by the application.

Bug 3 – Synchronization Error: A synchronization error occurs when the logic and the configuration of the application are correct, but the desired result is not produced because of message loss, data races, or asynchronous execution. For example, if node 1 has the maximum sensor reading in its neighborhood and node 2 has the second highest reading, node 1 should be elected the leader and node 2 should not. If node 2 does not receive the cache update message with node 1's sensor reading, however, the local computations on node 2 would compute that it has the highest reading and it would also be elected the leader. This data synchronization bug would produce the manifestation of having two leader nodes in the same neighborhood. In a similar scenario, node 1 may read from its sensor before receiving the cache updates from any of its neighbors, perhaps due to network delays or because node 1 sensed much earlier than its neighbors. Its local computations would therefore observe that its neighborhood was not active, and it would not be elected the leader. Meanwhile, all of node 1's neighbors would conclude that node 1 was the neighborhood leader because it has the highest sensor reading. This bug would manifest in the application missing the detection of a mobile object.

3.3 The MDB User Interface

MDB is a post-mortem debugger, which means that the user steps through a pre-recorded execution trace and does not view distributed state at execution time. As a result, MDB is able to provide *time travel* commands, allowing a user to step both forward and backward through the code. MDB also provides *historical search*, which allows the user to search over the historical sequence of distributed states for the manifestation of a bug without manually stepping forward and backward. For example, the user can find the time that a variable had a particular value, or can plot all sensor readings from a particular node.

The disadvantage of post-mortem debugging is that the user cannot modify system state during execution, as one might when using an on-line debugger such as GDB [8]. MDB overcomes this limitation to some extent by providing *hypothetical changes*, which allows the user to view how certain changes to the program would affect system state. Hypothetical changes allow a user to test whether a particular change to a program would fix a given bug, without the need to redeploy, execute, and test the new code.

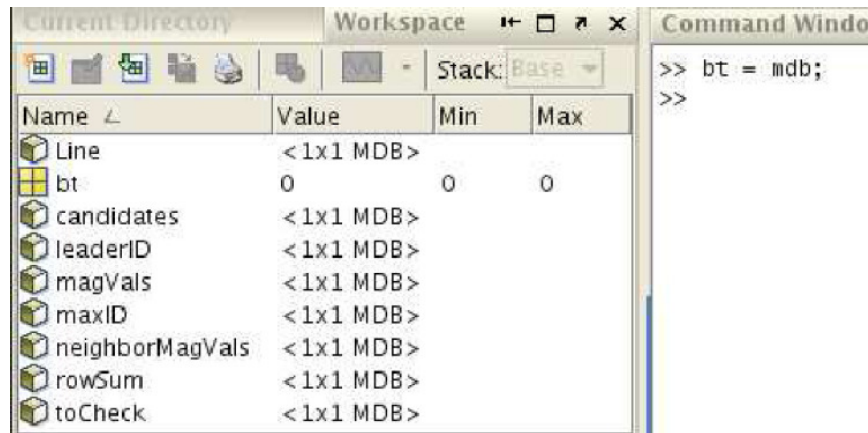


Figure 3.1: Starting the MDB debugger creates new objects in the Matlab workspace for every macrovector in the program being debugged.

While the prototype implementation of MDB targets the MacroLab programming environment, the principles behind the debugging approach can be applied to many different macroprogramming abstractions. Because MacroLab uses a Matlab-like syntax, the current implementation of MDB is designed to be used through the Matlab command prompt. To debug the execution of a MacroLab program using MDB, the user executed the `mdb` command which starts up Matlab with the workspace

initialized with the macrovectors in the program being debugged as shown in Figure 3.1. Table 3.1 provides a summary of the main commands offered by MDB.

| Command | Description |
|---------------------|---|
| tjump (t) | change the state of the system to time t μs |
| tstep $[(t)]$ | change the state of the system to next logged time [or current time + t] |
| lbreak (l) | place a breakpoint at line l |
| lstep $[(l)]$ | increment to the next line [or step l lines] |
| lcont | move forward to the next breakpoint |
| lstatus | list all breakpoints |
| lclear $[(l)]$ | remove breakpoint [at line l] |
| isCoherent (x, y) | check if x is coherent with y |
| diff (x, y) | compare views x and y of a vector |
| alt (hc, tl) | produce a timeline by altering tl using hypothetical change hc |
| getTime | get current debugger time |

Table 3.1: Basic commands provided by MDB. These commands allow the user to (1) navigate the trace temporally, (2) navigate the trace logically, (3) compare macrovectors, and (4) make hypothetical changes to the code.

3.3.1 Logically Synchronous Views

Logically synchronous views allow the user to inspect and analyze the distributed state of the system when all nodes are executing the same logical operation, (i.e., the same line of code.) MDB provides three commands for generating and navigating through logical views: **lbreak**, **lcont**, and **lstep**. These commands are used to debug logical errors such as Bug 1 from Section 3.2, where the system fails to detect moving objects because of a typo on line 8. The user wants to first inspect the sensor values stored in **magVals** and so places a breakpoint on line 8 using the **lbreak** command:

```
>> lbreak(8)
```

```
Breakpoint set at line 8
```

The user then executes the **lcont** command, which advances the state of the application on all nodes until just before they execute the operation on line 8 for the first time:

```
>> lcont
```

```
At Line 8
```

The programmer views the value of the **neighborMag** macrovector by simply typing the variable name:

```
>> neighborMag
```

```
neighborMag =
```

```
(1,1)    540
(1,4)    505
(1,7)    523
...
```

Since node 1 has at least three neighbors with sensor readings above **THRESH**, the programmer expects it to be in an *active* neighborhood. The programmer progresses past line 8 with the **lstep** command:

```
>> lstep
```

```
At Line 9
```

and views the value of the **active** variable:

```
>> active
```

This action reveals that **active** is an empty vector, even though node 1 has at least three active neighbors. At this point, the programmer inspects line 8 of the macroprogram and finds that the logic of the program was specified incorrectly: line 8 compares **THRESH** to the **magVals** variable instead of the **neighborMag** variable.

The key challenge to providing a logically synchronous view of macroprogram execution is that the nodes may take different paths through the macroprogram, but the system must still allow the user to step sequentially through the macroprogram. MDB allows this by using an intuitive forward ordering of program statements: when the user executes the **lstep** command, MDB only advances those nodes for which the next instruction has the lowest line number of all nodes' next instructions. More formally, **l** is defined to be the vector of line numbers l_i for each node i in the current logically synchronous view, and define **n** to be the vector of the next line numbers n_i that each node i executed and logged immediately after line l_i . When the user executes the **lstep** command, the next logically synchronous view will be the vector **l'** of line numbers l'_i , which are defined as

$$l'_i = \begin{cases} n_i & \text{if } n_i = \min(\mathbf{n}) \\ l_i & \text{otherwise} \end{cases}$$

Figure 3.2 illustrates the effect of intuitive linearization, where the solid lines show the sequence of statements that each node executes in the example program, and the dashed lines illustrate the ten logically synchronous views that MDB would create as a user executes the **lstep** command. In the first three views, all nodes progress through lines 1, 2, and 3 simultaneously. Node j iterates through

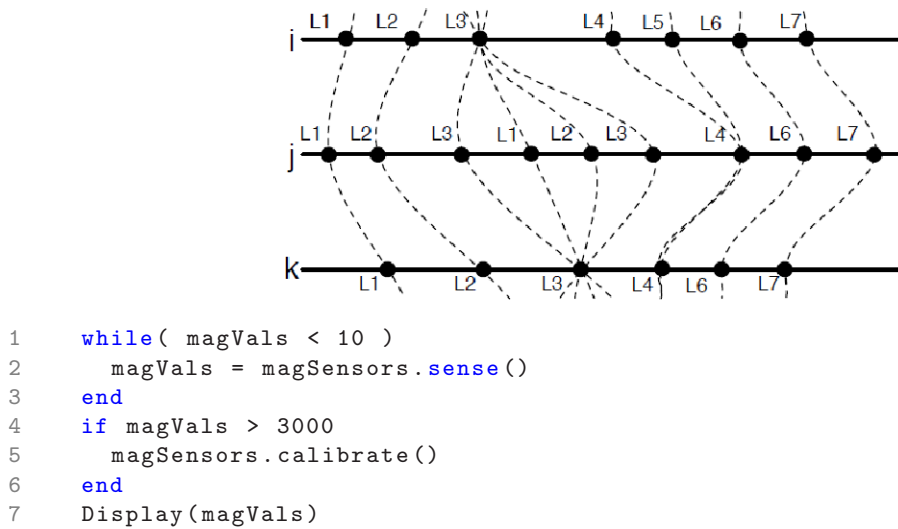


Figure 3.2: Logically synchronous views (shown by dashed lines) depict nodes i , j , and k progressing through the code in unison, even though node j executes the loop twice and only node i executes line 5.

the loop twice, executing lines 1, 2, and 3 again. In the logically synchronous views, nodes i and k remain at line 3 until node j exits the loop, at which point all nodes progress to line 4 together, as illustrated by the dashed lines. Similarly, nodes j and k remain at line 4 in the logically synchronous views until node i finishes executing line 5, when the views show all three nodes progressing to line 6 together. Thus, logically synchronous views depict all nodes progressing through the code in unison, even though they may take different paths through the code or execute the same lines of code at different times.

Logically synchronous views provide a convenient abstraction for navigating back and forth through the logic of a macroprogram, but they are limited in that the views of distributed state may include values that correspond to different points in time. This could lead to *causally inconsistent* states being presented to the user. For example, by the time node i reaches line L , node j may already have passed line L and sent i a cache update. Thus, a logically synchronous view at line L would show nodes i and j in a causally inconsistent state. Although all elements of the global state shown in such a view are not causally consistent, the view itself is still useful for debugging. In WENs, the user may also want to see distributed state at the specific time that a physical event took place in the network. For this reason, MDB also provides temporally synchronous views, described in the next section.

The technique described above for representing branches and loops in logically synchronous assumes that all nodes execute the same code and that no node fails or blocks indefinitely on a variable. The first assumption is inherent in the technique for stepping through non-sequential code sequentially and, therefore limits the usability of the logically synchronous views in MDB while the second assumption can easily be eliminated by monitoring for a node failure or indefinite block when generating the next logically synchronous view in response to an **lstep**. In the event that no node's state can be updated based on the technique above, yet all the nodes are not at the same macrocode line number, any node that is not at the common macrocode line can be assumed to have failed and be ignored in generating future logically synchronous views.

3.3.2 Temporally Synchronous Views

Temporally synchronous views allow the user to inspect and analyze the distributed state of the system at a specific time. MDB provides two commands for moving forward and backward in time: **tjump** jumps to a specific time, and **tstep** moves forward or backward by a given duration. These commands can be used to debug configuration errors such as Bug 2, described in Section 3.2, in which the mobile objects move by the nodes much more quickly than the rate at which they sample from the sensors. In this experiment, the programmer observes an object moving through the network approximately 0.4 seconds after execution started, but the object is not detected by the network. The programmer jumps to this time using the **tjump** command:

```
>> tjump(400000)

At 400000 microseconds
```

The programmer then inspects the **neighborMag** macrovector:

```
>> neighborMag

neighborMag =

    (1,1)    508
    (1,4)    125
    (1,7)    207
    ...
```

The user can see that only one node has a sensor value above **THRESH** at that time. Then the user steps forward in time by entering the **tstep** command:

```
>> tstep
```


At 404129 microseconds

When issued with no parameter, `tstep` continually steps forward to the next log entry. The programmer notices that node 1's neighboring nodes do not read from their sensors until long after the mobile object has left the vicinity, indicating that the sampling frequency is not high enough. This example illustrates how temporally synchronous views allow the user to view distributed state at the actual time a bug manifestation was observed. This is particularly useful in WENs, where bug manifestations may correspond to physical events that do not correspond to the logic of the macrocode.

Temporally synchronous views allow the user to navigate through the execution trace of a WEN, but are limited by the fact that the user must be able to specify the exact times of interest. Identifying the time that an interesting event occurred to within milliseconds or microseconds can be a challenge, especially when the timestamps do not necessarily correspond to the values of any given wall clock. For this reason, MDB also provides capabilities for *historical search*, as described in the next section.

3.3.3 Historical Search

Historical search allows the user to inspect and analyze the historical sequence of distributed system states without manually stepping forward and backward through the code. Like most source-level debuggers, the temporally synchronous and logically synchronous views only show the user one state of the system at a time. This requires the user to step forward and backward through execution while trying to remember and correlate values from different states to find the cause of a fault. Historical search allows the user to operate over the state of the system at all times at once, eliminating the need to step forward and backward in time.

MDB allows the user to access the history of a macrovector by adding a new dimension to it which represents time. For example, in the object tracking application in Figure 2.8, `magVals` is a single-dimension macrovector that is indexed by node ID. The value of `magVals` at node 5 and time 1000 can be accessed with the command:

```
>> magVals(5, 1000)

ans =

    17316
```

Macrovectors conform to standard Matlab syntax even with the additional time dimension, and standard Matlab operators can be applied to them. This produces a powerful programmatic debugging

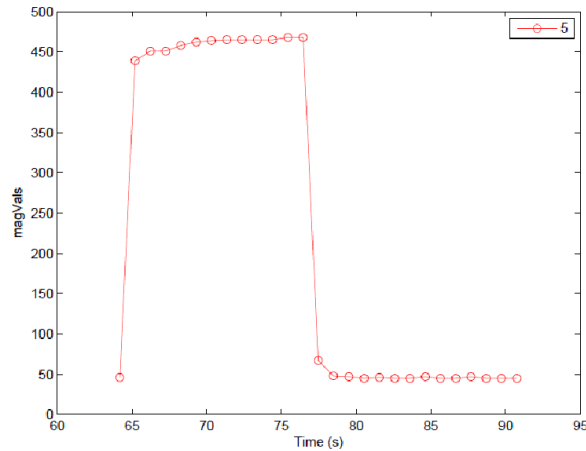


Figure 3.3: MDB’s views of distributed state can be combined with Matlab’s rich plotting and analysis tools. This figure is the result of a single debugging command: `plot(magVals(5, :))`

interface that can be used to search for the manifestation of bugs. Historical search can be used to find timestamps of all instances of Bug 3 in which more than one leader was detected, using a single command:

```
>> find(numel(leaders(:, :)) > 1)

ans =

    17317316
   39824496
```

where `numel` is a standard Matlab operation that returns the number of elements in a vector. In this scenario, the developer finds that the IDs of more than one node were stored in the `leaders` macrovector at two times during program execution. To identify the cause of the bug, the programmer then jumps to one of these times using the command:

```
>> tjump(17317316)

At 17317316 microseconds
```

MDB’s historical search functionality allows the user to exploit Matlab’s rich plotting and analysis tools. For example, the command

```
>> plot(magVals(5,:))
```

produces the graph shown in Figure 3.3 that contains all sensors values read by node 5. Historical search provides a convenient mechanism for searching through and analyzing the historical sequence of distributed system state.

3.3.4 Hypothetical Changes

MDB allows the user to observe how *hypothetical changes* to a macroprogram would affect distributed state, without redeploying and executing the new code. This functionality is useful for testing whether a particular change will fix all occurrences of a bug that appeared in a given execution trace. As a proof-of-concept, MDB currently provides four hypothetical changes that highlight the effects of adding process and data synchronization primitives to a macroprogram: (i) a hypothetical barrier (ii) hypothetical cache coherence (iii) a hypothetical time delay, and (iv) hypothetical cache expiration.

The original execution trace that was collected during program execution is called the *base timeline* and is denoted *bt*. Hypothetical changes are applied to the base timeline to generate an *alternative timeline* denoted *at*. For example, the programmer could make a hypothetical change to a program by setting `magVals = 0` at line 7 by executing the `alt` command:

```
>> at = alt('magVals = 0', 7, bt)
```

This command would produce *at* by modifying *bt* such that `magVals` is always set to 0 at line 7. The value of a macrovector in this alternative timeline can be viewed by passing a handle to the timeline as an optional last parameter when indexing that macrovector. For example, a user can view node 7's value of `magVals` at time 10000 in the alternative timeline by executing the commands:

```
>> magVals(7, 10000, at)
```

Hypothetical Barrier

A *barrier* is a point in the source code that all nodes must reach before any node can proceed. The *hypothetical barrier* illustrates how the distributed state of the system would change if a barrier were placed at a particular line of macrocode. For example, the user can use hypothetical barriers to test whether a barrier at line 12 in the example program from Figure 2.8 would fix Bug 3 from Section 3.2. To do so, the programmer first creates an alternative timeline with a hypothetical barrier at line 12 using the command:

```
>> hb = alt('barrier()', 12, bt)
```

The user then checks if multiple leaders still arise with a barrier by apply the same command that was used in Section 3.3.3 to the new timeline:

```
>> find(numel(leaders(:, :, hb)) > 1)
```

Because this command no longer returns any timestamps, the user concludes that applying a barrier to line 12 would fix Bug 3. Thus, this hypothetical change allows the programmer to test whether this code modification, with respect to the given trace, fixes all occurrences of this bug, some occurrences, or no occurrences, without the need to redeploy, execute, and test the code.

The semantics of a hypothetical barrier is illustrated using the example execution timelines shown in Figure 3.4(a). MDB first creates a logically synchronous view at line 12, as illustrated by the dashed line. If the nodes had implemented a barrier, the state of the system would be identical to the state represented by this logically synchronous view, except that additional cache update messages would have been received. Specifically, any node a that waited at the barrier would have received all messages from another node b that were sent before b reached line 12 and that were received before all nodes reached line 12. Thus, MDB creates the hypothetical barrier by updating the state of the logically synchronous view at line 12 with all such cache update messages. This hypothetical change to the order in which cache update messages are received is illustrated by a dashed arrow in Figure 3.4(a) to illustrate the difference between the messages that were received in the real execution trace bt , and the messages that were received in the alternate timeline at . The distributed state resulting from this message re-ordering is the view of a hypothetical barrier applied to line 12.

Hypothetical Time Delay

Barriers are expensive operations and are not always desirable in WENs due to their high message cost. A cheap alternative is to have all nodes wait for any cache updates to arrive for only a fixed period of time before continuing execution. If execution on all nodes is already synchronized or is otherwise known to be unsynchronized by a bounded duration, this approach may provide sufficient data synchronization guarantees. The *hypothetical time delay* produces the distributed state that would have been produced if a time delay of Δt were inserted at a particular point in the macrocode. The programmer could create a hypothetical time delay using the `deltat` command:

```
>> dt = alt('deltat(10)',12, bt)
```

`deltat` takes a parameter to indicate the delay Δt in milliseconds. To create the hypothetical time delay, MDB first creates a logically synchronous view on line 12. Then, the distributed state is updated with any cache update messages received by a node within the Δt of the time it reached line 12. Figure 3.4(b) illustrates that a hypothetical time delay applied at line 12 would cause the

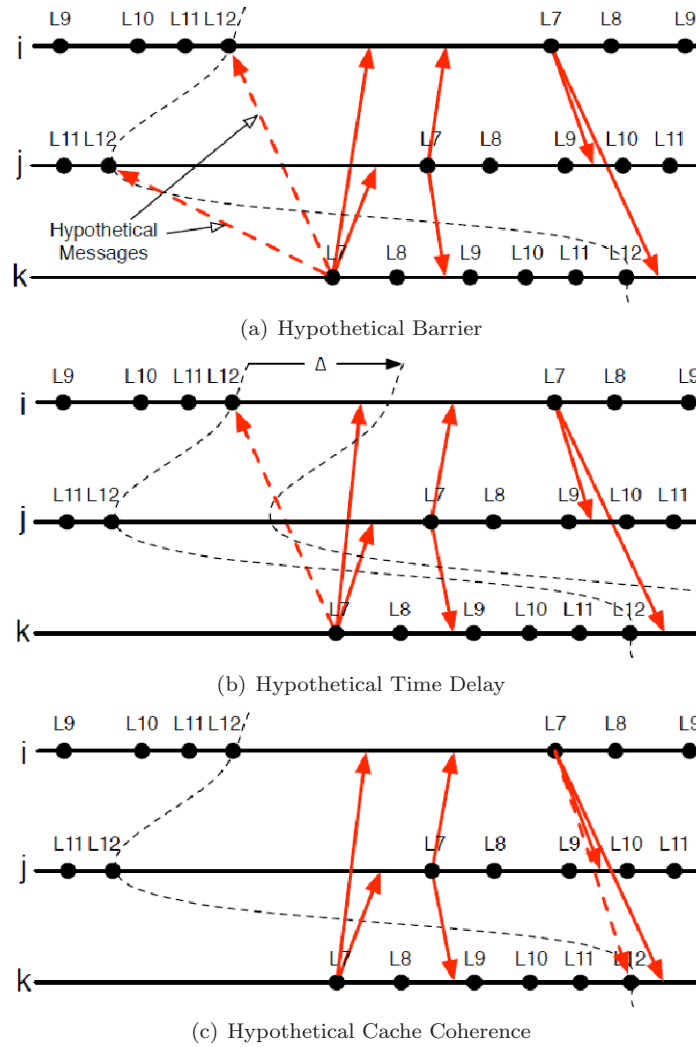


Figure 3.4: MDB emulates the effect of hypothetical code changes by reordering messages appropriately. Solid arrows illustrate message reception in an actual execution trace. Dashed arrows illustrate how these messages are re-ordered to emulate the effect of hypothetical changes to the code.

message sent from line 7 of node k to update the state of node i after executing line 12.

Hypothetical Cache Coherence

Hypothetical cache coherence shows the hypothetical distributed state if all caches were coherent at a given time. This view can be produced with the command:

```
>> cv = alt('coherent()', 12 bt)
```

To create hypothetical cache coherence, MDB rearranges any cache updates that were sent before,

but received after the line of code specified. Figure 3.4(c) illustrates how message reception is altered by the command above: the message from node i is received at node k on line 12, because it was sent before k finished executing line 12. Perfect cache coherence is expensive to implement because it requires two-phase locks to be acquired on all cached versions of a variable before any new values can be written to it. If hypothetical cache coherence makes a bug manifestation disappear, the user may attempt to use end-to-end reliability for cache updates, low-latency communication protocols, or other mechanisms that will improve, but not guarantee, cache coherence.

Hypothetical Cache Expiration

Cache expiration is the process of invalidating a cached copy of a value once it becomes too old. This is cheaper to implement than perfect cache coherence, but it also provides few correctness guarantees because the original value may change before the cached copies of the old value expire. Furthermore, when nodes run asynchronously and aperiodically in a network, it can be difficult to decide when to expire cache entries. Expiring too quickly can result in insufficient data, but expiring too slowly can result in the use of stale values. *Hypothetical cache expiration* shows the hypothetical state that would result if cache expiration were used at a given line of code, with a particular expiration time. This view helps the user evaluate the effect of different expiration times and can be created using a command such as:

```
>> ev = alt('expire(100000)', 12, bt)
```

The values in the view are produced by finding the last value written to each element of the macrovector in the time interval $[t - t_e, t]$ where t is the time the node executed line 12 in the current instance of the logically synchronous view and t_e is the expiration time, in this case $100000 \mu s$. If an element has had no value written to it within that time interval, the resulting view has a *NaN* value for that element.

3.4 The MDB Implementation

This section describes the implementation of MDB, including the collection of execution traces (Section 3.4.1), how MDB Lite reduces energy cost (Section 3.4.2), how timekeeping is performed on data traces (Section 3.4.3), how MDB's global views and historical searches are implemented using

these data traces (Section 3.4.4), and how hypothetical changes are emulated by creating modified copies of these data traces (Section 3.4.5).

3.4.1 Creating Execution Traces

MDB is a post-mortem debugger, which means that it must collect execution traces at run-time that will be analyzed after execution is complete. Most post-mortem debuggers create *event traces*: they log all non-deterministic events such as interrupts, I/O, and messages. These event traces are sufficient to recreate the state of the system at any point of the original execution using *execution replay*. In contrast, MDB creates *data traces*: it logs all changes to the system state, including writes to variables and changes to the program counter. Event traces generally produce shorter logs than data traces because events are relatively rare while state changes occur with every line of code executed, especially for traditional distributed applications that are compute intensive but have relatively little I/O or network communication. In contrast, each line of a macroprogram may correspond to many machine instructions, I/O operations, and network events. Therefore, data traces are more efficient than event traces for logging macroprogram execution.

MDB log entries consist of the (macro)program counter, the variable location and value being written, a 48 *bit* microsecond-accurate timestamp, and the sequence number. The sequence number helps correlate variable write events with remote cache events of that same value, since message transmission and reception are not logged. After execution is complete, the logs are retrieved from the nodes using the collection tree protocol from the TinyOS source tree, although any data collection protocol can be used.

To minimize interference with the application, MacroLab logs data in two phases: first in RAM and then to external flash. While the MacroLab application is executing, trace entries are stored in a circular RAM buffer, requiring only 105 machine instructions per log. Then, when the node has no more instructions to process, and right before it enters sleep mode, the RAM buffer is dumped to external flash, which takes approximately 50 ms. This approach reduces MDB's interference with the application by reducing contention for the CPU. One disadvantage of this approach, however, is that any log entries in the RAM buffer may be lost if the node crashes. All logging code required for MDB is automatically added by the MacroLab compiler when the debugging option is enabled.

The external flash is 1 *MB* in size, and when it is full the earliest trace entries are overwritten. One advantage of MDB's use of data traces is that old trace entries can be overwritten when the

external flash is full. In contrast, entries in an event trace cannot be overwritten because all events from the beginning of execution are required for execution replay. Therefore, systems that collect event traces must use checkpointing techniques, which can introduce additional overhead [72].

3.4.2 MDB Lite

MDB Lite is a light-weight version of MDB that conserves energy by not writing log entries to the external flash, although all other logging code is included in the microprograms and is identical to MDB. Furthermore, MDB Lite emulates the process of writing to flash by using a hardware timer to turn off the appropriate interrupts for the appropriate period of time. The logging commands and flash emulation ensure that the timing and memory characteristics of a program are the same when executing MDB and MDB Lite. Thus, MDB Lite cannot be used for debugging, but it can be enabled during the deployment phase to reduce energy overhead while also eliminating the possibility of heisenbugs: a change in the manifestation of bugs when the debugger is enabled or disabled. The user can toggle between MDB and MDB Lite to enable or disable debugging.

3.4.3 Distributed Timekeeping

After a program executes and the logs are collected, MDB must ensure that the timestamps in the logs are *causally consistent*: any event e that causes event e' must have a smaller timestamp than e' . Distributed time keeping is a challenging problem because nodes do not share a common clock, and is the main distinguishing feature between debugging on a distributed system and debugging concurrent threads on a single machine or a shared memory multiprocessor (SMMP). MDB adopts a well-known approach that is sometimes called the *Lamport* algorithm: all logs are timestamped with a value from the node's microsecond counter, and clock skew is accounted for by enforcing that a receiver's local time when a message is received is greater than the sender's local time when the message was transmitted. Since message passing is typically the only form of interaction between nodes, this approach guarantees that any events on the sender have an earlier timestamp than events they might cause on the receiver. Any distributed timekeeping scheme that satisfies this property is said to support *Lamport time* [85]. Other timekeeping schemes such as vector clocks have more precise causal semantics [86, 87]. MDB's use of the Lamport algorithm occurs off-line after the logs are collected, which eliminates the need for any run-time overhead due to on-line time

synchronization [88, 89]. One disadvantage of this approach is that all nodes must receive periodic messages from neighboring nodes. In a partitioned network where some nodes do not have radio connectivity with other nodes, the logs cannot be made causally consistent. This is the case for all distributed systems, and not a limitation specific to MDB.

Distributed timekeeping in WENs is complicated by the fact that nodes can also interact through their sensors and actuators: two nodes can sense the same stimulus, or a node's sensor reading can be affected by another node's actuator. This causal relationship is not explicit in the program, cannot be verified at run time, and cannot be enforced by the Lamport timekeeping algorithm. Such node interactions through events external to the synchronization messages is called *anomalous behavior* by Lamport, who suggested physical clocks to eliminate this limitation. Empirical studies of the CPU clocks are performed on the Tmote Sky nodes and found an average clock skew of $139 \mu\text{s/s}$, similar to the observations mentioned in RBS [88]. The inherent uncertainty of sensor readings on the same nodes is measured to be about 20 ms, which is the average time required for the ADC to digitize the analog readings from the sensors, plus software overhead. Thus, a minimum message rate of about one message every 2 minutes is required to prevent clock skew from growing larger than the inherent uncertainty on sensor reading timestamps. The minimum message rate may even be much larger when measuring physical stimuli such as temperature or object mobility that change with periods much lower than 20 ms.

3.4.4 Generating Global Views

Once the execution traces are retrieved and synchronized, MDB can use them to create views of any system variable at any given time. To view the value of variable x at time t , MDB indexes the trace of the node on which variable x was logged and retrieves the last value written before time t . For example, to produce the value of node k 's element of the `magVals` vector at time 1000, the system may retrieve the value that was written to `magVals` by node k at time 998. This basic functionality helps implement MDB's logically synchronous views, temporally synchronous views, and historical search.

Logically synchronous views are implemented using an `lline` vector which stores all the line numbers at which a breakpoint has been placed and an `ltime` vector which stores a time for each node to indicate MDB's location in its trace. When the user enters `lbreak` with a line number, the line number is appended to `lline`. When the user enters `lstep`, MDB identifies which nodes

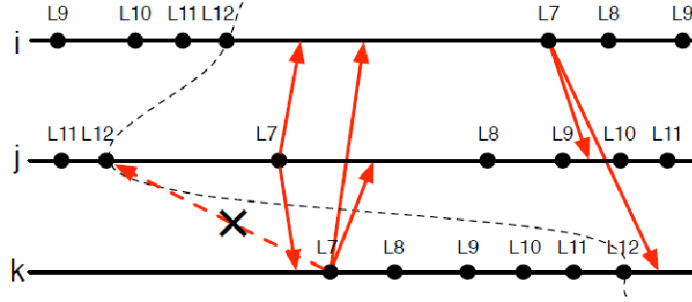


Figure 3.5: A hypothetical barrier cannot be applied in this scenario because the hypothetical advancement of the message from node k to node j would create a dependence cycle.

executed the next line of the macroprogram next and sets the value of `ltime` for those nodes to be the time when the node finished executing that line. When the user executes `lcont`, MDB executes `lstep` repeatedly until the macrocode line matches one of the values in `lline`.

Temporally synchronous views are implemented using a variable called `ttime` which stores the parameter passed to `tjump`. `ttime` is updated when the user issues the `tstep` command. If the user views a macrovector after `ttime` has been set, the view of the macrovector is generated by searching through the trace of each node for the last state update to the macrovector at or before `ttime`. Historical search is implemented by searching through the execution traces for all writes to the macrovector specified by the historical search.

MDB uses a special *NaN* value to represent any value in a logically synchronous or temporally synchronous view that does not exist in the logs. This situation occurs in a temporally synchronous view, for example, when a node fails or stops executing and the log from that node does not contain values beyond a certain point in time. It may also occur in logically synchronous views, for example, if a node blocks on a variable indefinitely or dies and its log has fewer instances of a particular line number than the logs of other nodes.

3.4.5 Generating Hypothetical Changes

Hypothetical changes are made by modifying an initial timeline denoted `it` to generate an alternate timeline denoted `at`. This process involves two phases: (1) applying hypothetical changes to `it`, and (2) propagating the hypothetical changes into the future. MDB carries out the first stage of alternative timeline generation by copying all the values from `it` into a new timeline `at`. Then, it generates logically synchronous views at each instance of line l specified by the user and MDB modifies the

cache update times based on the hypothetical change specified, as described in Section 3.3.4. For example, to implement the hypothetical barrier depicted in Figure 3.4(a), MDB would reorder the trace entries in the new timeline **at**, such that the message from line 7 of node k is received by nodes i and j when they reach the barrier. After the message reordering is complete, MDB propagates this change into the future by re-executing the instructions corresponding to every subsequent log entry in **at**, in the order of their timestamps. Thus, MDB re-executes the instructions logged in the original execution trace to propagate the state from the hypothetical change into all future states. After re-executing each line of code, MDB re-creates a new trace entry and adds it to the alternative timeline **at**. Thus, alternative timeline **at** should have exactly the same timestamps on all log entries as the original timeline **it**, but might have slightly different values.

When a node reads from a sensor, MDB retrieves the sensor reading that was collected in the original execution trace. MDB does not attempt to generate simulated sensor values based on a model of the stimulus. Therefore, creation of the alternative timeline cannot proceed once the control flow of execution diverges from the control flow observed in the original execution trace. Thus, hypothetical changes can only be propagated into the future insofar as they do not change control flow; any request for a view of the alternative timeline after control flow changes produces the *NaN* value, to indicate that the view cannot be generated. This limitation could be overcome by incorporating simulation and sensor models into MDB, but this is beyond the scope of this dissertation.

In addition to control flow changes, alternative timeline generation also fails in certain instances when the user specifies a hypothetical change that results in a *causality cycle*: an event e that is caused by event e' reordered to occur before e' . For example, in Figure 3.5, the hypothetical barrier applied to line 12 would cause message m from line 7 of node k to be received before message m' is sent at line 7 on node j . However, this produces a causality cycle because message m' is also received by node k before it sends message m . This scenario is called the *grandfather paradox*. MDB checks for the grandfather paradox during the creation of alternative timelines, and all subsequent values that are causally related to m or m' are assigned the *NaN* value.

3.5 Macroprogramming Properties for MDB

While MDB is currently implemented for MacroLab, its principles apply to other macroprogramming systems. This section defines the language and system properties necessary to support various aspects of MDB and identify other macroprogramming systems that share these properties.

High-level abstraction. The high-level programming abstraction of MacroLab concisely describes complex system states in a small number of lines of code and variables. This allows for efficient logging on resource-constrained devices. All macroprogramming abstractions, by definition, allow for such a succinct descriptions [5, 19, 90–92].

No message passing. MacroLab forbids user-written message passing, instead managing all communication in the run-time system. MDB can thus log only cache update messages. Systems such as Hood [12], Pleiades [19], COSMOS [93], and Semantic Streams [7] also abstract away message passing from the programming model.

Clear mapping from macro-code to micro-code. This is essential to log the ends of macrocode lines in microcode. Most macroprogramming systems, such as [19] translate the macrocode statements into appropriate blocks of microcode and therefore possess this property.

In-order message delivery. This enables cache updates from a given node to be received in the order they occurred in locally. Most macroprogramming systems are built on this assumption [94]. If not, sequence numbers can be used to prevent out-of-order cache updates.

Sequential imperative language. MacroLab allows logically synchronous views to be generated where the user can place breakpoints and step through code. This property is supported by all sequential imperative macroprogramming systems, such as Marionette [18].

Data caching. Data caching, in the form of neighbor reflections, yields Lamport time stamps without additional logging. This property also makes hypothetical views both possible and useful since they show how data caching would be affected by application changes. All macroprogramming abstractions that support data caching, such as Hood [12], provide this property.

Single machine abstraction. MDB displays the state of the system as the state of the macrovectors if they were centralized. This requires the system to provide a single machine abstraction to the network. Systems such as RuleCaster [16] also possess this property.

Data-centric. The data-centric nature of MacroLab enables data traces, rather than event traces, to be logged. Semantic Streams [7], COSMOS [93], and TinyDB [4] are similar data-centric

languages.

3.6 Evaluation

MDB is evaluated in two phases. First, it is shown that data traces are more efficient for macrodebugging than the traditional approach of creating event traces. Then, the RAM, flash, energy, and CPU cycle overhead of MDB is evaluated.



Figure 3.6: MDB is evaluated on three macroprograms by running them on a 21 node testbed. Light sensors and a projector created the sensor stimuli.

3.6.1 Experimental Setup

The evaluations were carried out on a testbed with 21 TMote Sky nodes with photoresistor sensors (Figure 3.6). Since all the nodes in the testbed are within one hop of each other, their communication ranges are artificially limited by modifying the packet reception module of the CC2420 radio. Nodes are restricted to communicate with neighbors next to them vertically, horizontally, and on the two diagonals. To collect additional data that could not be obtained from the testbed, the node-level Cooja network simulator [95], which makes use of the MSPSim [96] TMote Sky simulator, is used.

```

1 every(uint16(1000))
2   trigger = microphone.sense();
3   meanTrigger = mean(neighborTrigger, 2);
4   candidates = find(meanTrigger > THRESHOLD);
5   soundLog(candidates) = microphoneHF(candidates).sense();
6   baselog(soundLog);
7 end

```

Figure 3.7: The acoustic monitoring application reads values from a microphone and reads from a higher-fidelity microphone on nodes whose neighborhoods detect high average noise levels.

The microcode executing on the nodes in the Cooja simulator is exactly the same microcode that executes on the real Tmote Sky hardware.

MDB is evaluated with three macroprograms. The first application is PEG, described earlier (Figure 2.8), for which the movement of objects is emulated using the light sensors by moving a white circle on a black background that was projected from a laptop onto the testbed. The intensity of the circle decreases radially outward to emulate the effect of varying sensor readings by nodes that detect the object. The second application is Surge (Figure 2.7), which is a simple data collection application that periodically samples from a sensor and displays the readings at a base station. The third is an acoustic monitoring application (Figure 3.7), which first senses from a microphone sensor, and depending on the number of neighbors that also heard a sufficiently loud sound, samples from a second high-fidelity microphone, stores the values, and reports them to the base station. The sensor values for both of these applications are generated using the photoresistors on the nodes.

3.6.2 Data vs. Event Logging

The cost of creating data traces is evaluated by counting the number of logging statements required to record all variable writes and program counter changes for a single run of each of the three macroprograms. This count is compared to the number of interrupts that would need to be logged to create an event trace of the same program execution. Since the interrupts generated on real hardware could not be counted without changing the timing characteristics of the program, these values were obtained by analyzing the programs as they executed on the Cooja network simulator. Figure 3.8 shows that, for these applications, the number of hardware interrupts is 14–32 times larger than the number of updates to macroprogram state. These results clearly show that MDB’s approach of collecting data traces is much more efficient for macroprograms than the traditional

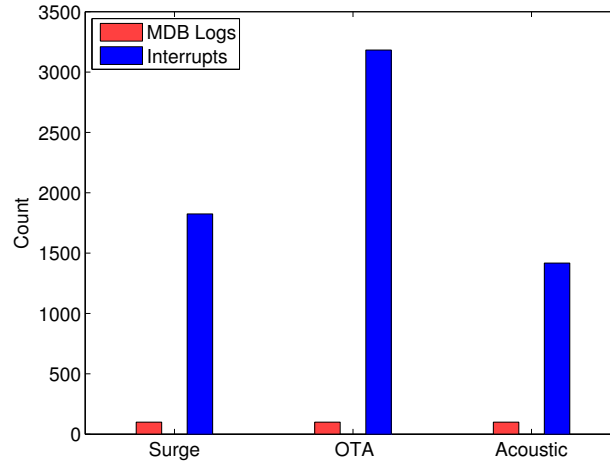


Figure 3.8: Number of interrupts generated per 100 data states written to flash. WEN applications produce 14–32 times more interrupts and message events than macroprogram state updates.

approach of collecting event traces.

3.6.3 RAM and Flash Overhead

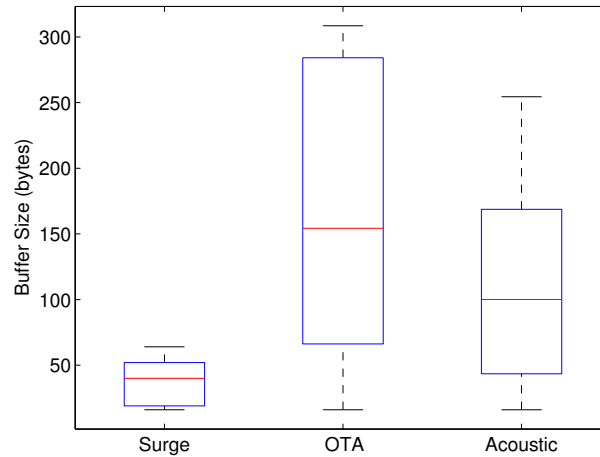


Figure 3.9: Log data is stored temporarily in RAM before being written to flash. In the experiments, no node needed more than 304 Bytes of RAM. The box plot shows the minimum, lower quartile, mean, upper quartile, and maximum values.

Memory is not typically a concern for traditional debuggers that execute on PCs, but WENs are composed of highly resource-constrained devices and efficient memory usage is essential to making

| Application | Flash (<i>Bps</i>) | Wraparound (<i>hr</i>) |
|-------------|----------------------|--------------------------|
| Surge | 31 | 9 |
| Accoustic | 187 | 2 |
| PEG | 288 | 1 |

Table 3.2: Flash memory consumption is low enough to debug hours of execution. The buffer is circular, so it can always store data to debug the last few hours of execution.

MDB practical in this domain. The amount of memory MDB requires is measured by instrumenting the RAM buffer portion of the logger and tracking the maximum difference between the head and tail pointers of the circular buffer. Figure 3.9 depicts box plots that show the minimum, maximum, mean, and the lower and upper quartiles of RAM consumption for each of the three test applications executing on a 21 node testbed. This data reveals that MDB has modest RAM requirements, and needs to store a maximum of 304 *Bytes* of data, while the Tmote Sky nodes have about 10 KB of RAM available.

The amount of Flash memory required by MDB to store the complete data traces for each of the three applications was measured. Table 3.2 shows that the applications store less than 300 *Bps* to the flash. At these rates, simple applications such as Surge can collect logs for several hours before exhausting the 1 *MB* of external flash available on the Tmote Sky. More complicated applications such as PEG and acoustic monitoring may exhaust the flash after about one and a half hours. Thus, bugs in these applications must be detected within one and a half hours in order to debug them before the log entries are overwritten.

3.6.4 CPU Overhead

The effect of MDB on the execution speed is evaluated by counting the logging instructions executed during a particular run of the three applications. Since it was difficult to collect this information from the actual nodes without substantially modifying the timing characteristics of the application, this data was collected by running the application on the Cooja simulator. Since Surge has no interaction between nodes, it is simulated using 1 node. Acoustic monitoring used 5 nodes and PEG used 10 nodes. Figure 3.10 shows the number of MacroLab application instructions and MDB logging instructions that execute, as a percentage of the total execution time. The values in Figure 3.10 are averages over all the nodes simulated in Cooja. As Figure 3.10 shows, the MacroLab application code executes for between 2% and 6% of the total execution time and the logging code executes for

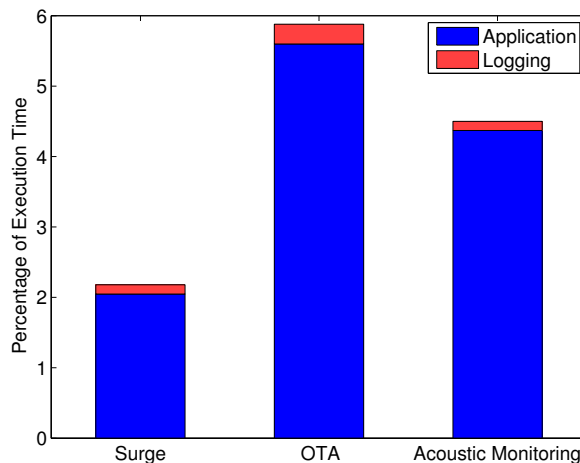


Figure 3.10: MacroLab context constitutes less than 6% of all code running on the nodes. Logging code constitutes less than 0.5%.

less than 0.5% of the total execution time.

It takes exactly 105 machine instructions to log data to the RAM buffer. Execution is minimal because on the MSP430 an instruction takes approximately 125 nanoseconds. This means the logging overhead is about 13 microseconds. This is extremely small when compared to the normal duty cycle of applications written in MacroLab.

3.6.5 Energy Consumption

MDB's energy consumption is evaluated by executing the PEG application, with the sensor being sampled every 10 seconds, on the 21 node testbed and measuring its average current draw over a period of 100 seconds. This experiment is executed with MDB, MDB-lite, and without MDB. The experiment is repeated both with and without low-power listening (LPL) [36]. Figure 3.11 shows that an application consumes 30% more energy with MDB, when LPL is enabled with a sleep interval of 1 second. This energy overhead is due to the process of saving logs to the external flash chip. With MDB Lite, this overhead is reduced to only 0.2 *mW* or 0.9%. This is because MDB Lite does not write to flash, and the timer-based implementation of MDB Lite allows the node to sleep when possible. The overhead of MDB and MDB Lite when LPL is disabled is only 0.5 *mW* because the node does not go into sleep mode in any of these implementations.

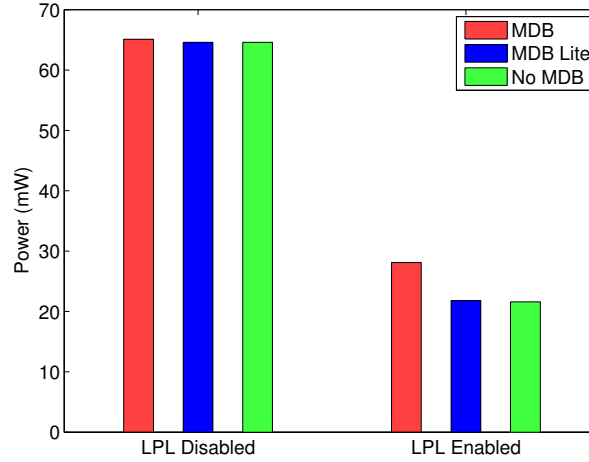


Figure 3.11: Without low-power listening enabled, applications consume 30% more energy with MDB enabled, but only 0.9% more energy with MDB Lite.

3.6.6 Discussion

The effectiveness of MDB as a CPS debugger depends on two factors: (i) the usefulness of the debugging abstraction and (ii) the efficiency of trace logging. The first factor is addressed by demonstrating that the features of MDB can be used to debug three classes of bugs in macroprograms (Chapter 3.3). MDB, by its very nature, does not enable the debugging of bugs in layers below the application layer. For example, a user cannot identify a bug in the MAC layer or routing layer because MDB hides message passing between nodes. This abstraction of low-level details conforms to the abstraction provided by the macroprogramming system since one of the goals of MDB is to allow the user to program and debug at the same level of abstraction. Also, MDB relies on the guarantees of macroprogramming systems that the low-level libraries have been rigorously tested and can be used in macroprograms to build more complicated applications. If a user of a macroprogramming system does not have such a guarantee about the function libraries, macroprogramming would become more of a burden on the user than writing the functions from scratch as the user would have to test the interoperability of various functions, that have been written by others, before they can be used to write an application. Thus, a debugger that allows the user to debug at the application level of a macroprogramming language is a useful debugging abstraction.

In order for MDB to be useful, execution traces should be collectible for a sufficient duration. MDB attempts to maximize the duration of execution for which traces can be collected by minimizing

the number of trace entries that have to be logged per unit of time. This is done by exploiting the fact that the user is debugging at the macroprogram abstraction and, therefore, only changes that are reflected in the macroprogram need to be logged. Thus, the many events, such as interrupts, that take place in the TinyOS operating system for instance can be ignored since they are not visible at the macroprogramming abstraction. What MDB does log are the writes to variables declared in the macroprogram a macroprogram counter that indicates which part of the macroprogram each node is executing. This enables MDB to log 14–32 times fewer trace entries per unit of time than if MDB were logging all events within a node so as to recreate each node’s execution.

MDB utilizes a temporary buffer in RAM to which macroprogram state changes are logged while each iteration of the macroprogram’s every loop is executing. At the end of each iteration, this buffer is written to flash. The amount of RAM required for MDB depends on the number of variables in the macroprogram, the number of lines of code in the macroprogram, and the number of neighbors with which each node shares macrovectors. The evaluations indicate that for PEG, which has 13 lines of macrocode with five vectors, on the 21 node testbed where each node has eight neighbors the maximum amount of RAM required is 304 *Bytes*. This is just 3% of the RAM available on Tmote Sky nodes. While larger, more complicated applications could require a greater amount of RAM for the buffer, the fact that PEG, which is one of the largest and most complicated WEN applications to data [12], requires such a small fraction of the available RAM indicates that MDB would not pose much of an overhead on the available RAM for most applications.

For the duration of the data collection phase the traces are stored on the node’s external flash. For the object tracking application traces for one hour of execution can be collected before the flash buffer gets overwritten. This duration can be increased for debugging purposes by reducing the sampling frequency or network density which would decrease the number of state changes that have to be logged. For most applications which would fall in the complexity spectrum between Surge, which is one of the simplest WEN applications, and PEG, around five hours of execution traces could be collected without overwriting the flash. For most WEN applications, five hours of execution should be sufficient during the debugging phase to analyze their performance and identify bugs. If the user needs data from a longer execution of the application, the traces can be collected periodically, just before the flash buffer gets overwritten, and the application could be allowed to continue executing.

Due to the controlled conditions under which the testing phase of application development usually

takes place, the limitations placed by the flash capacity can be overcome as described above. Yet, if the user wishes to deploy the application with logging enabled so that any bugs that arise during deployment can be analyzed, the flash capacity is of greater concern. For example, the presence of a bug in a data collection application may only be noticed when the data is analyzed at the end of a deployment. MDB may not be usable for identifying this bug if the bug occurred before the last n hours of execution if the flash buffer fills up every n hours. Yet, if the bug manifested itself within the last n hours, the available data traces could still be used with MDB to identify the bug. This would not be possible if MDB logged execution traces without resorting to checkpointing or other techniques for storing the state of the system periodically.

MDB impacts the execution speed of applications by consuming CPU cycles to carry out the logging operation. As Figure 3.10 shows logging to RAM imposes only a 0.5%, or $13\mu\text{s}$, overhead on the application execution time. Since the inherent uncertainty of even a sensor readings is an order of magnitude greater at 20 ms (Section 3.4.3), the $13\mu\text{s}$ overhead of logging can be ignored for almost all applications.

MDB also consumes energy while logging which could decrease the lifetime of the application. As Section 3.6.5 describes, an application with logging enabled consumes 30% more energy than the application without logging enabled. This is a considerable amount of energy since it could decrease the lifetime of a system by 30%, for instance causing an application that could have executed for a year to die in eight months. A user could mitigate this issue in two ways. The first is to increase the power available to each node by at least 30%. This can be done by providing each node larger or additional batteries or augmenting each node with energy scavenging capabilities. This would enable the full logging capabilities of MDB to exist while the application is executing in the real world so that if a problem were to arise the user could obtain the traces and attempt to identify its cause.

An alternative is to deploy applications in two phases. The first is a debugging phase where the application, with MDB logging enabled, is deployed for a short period of time so that traces can be collected and bugs identified. Then, after the identified bugs have been fixed, the application is deployed for its full duration without the MDB logging capabilities enabled so that there is no energy overhead. This is ideal if the application does not need logging during execution or if it providing each node with additional power is not feasible and the application cannot tolerate the energy overhead of MDB. Yet, an application without MDB logging enabled could behave differently

from an application with MDB logging enabled due to the difference in timing between the two. This is mainly the time MDB takes to write the RAM buffer to flash which could be about 10 ms per log entry. This timing difference could cause a bug that was hidden when MDB was enabled to manifest itself during the actual deployment. Such bugs are called Heisenbugs. MDB allows a user to deploy an application without flash writing enabled while still maintaining the timing characteristics of MDB by enabling MDB Lite instead of MDB when compiling the application. MDB Lite functions just like MDB except when writing the RAM buffer to flash. During this phase MDB Lite disables the interrupts which MDB disables when writing to flash and puts the node into a sleep state for the same duration of time MDB takes to write the RAM buffer to flash. This operation ensures that MDB Lite behaves in an identical manner to MDB in terms of timing, yet by eliminating the flash writes decreases the energy overhead from 30% to 0.9% which should be tolerable for most applications. Thus, as other projects such as [97] have claimed, MDB eliminates the possibility of Heisenbugs by enabling the tracing to be left on at all times.

3.7 Conclusions

Macroprogramming systems make programming a WEN easier by providing high-level distributed abstractions such as database tables, logical facts, or data streams so that the user does not need to build a mental model of the individual nodes and their interactions. However, no macroprogramming systems have debugging support, which is a crucial link in the development chain as a macroprogram progresses from the drawing board to real deployment. This chapter presents MDB, the first system that allows the user to inspect and analyze the execution of a WEN using the high-level abstractions provided by a macroprogramming system. This process, called *macrodebugging*, simplifies the debugging process and eliminates the need to analyze traces of low-level events and message passing algorithms. Macrodebugging is shown to be not only easier, but also more efficient than the debugging of node-level programs.

The MDB macrodebugger is built for the MacroLab macroprogramming system [98], but the underlying principles can be applied to other macroprogramming systems. The collection of data traces can be applied to any system that has a high-level abstraction for which the overhead of logging a single high-level operation is small compared to the number of low-level instructions required to execute that operation. This property holds for most macroprogramming systems. The source-

level debugging interface can be applied to any system that uses a sequential imperative language and has a clear mapping between macrocode and microcode, such as Kairos [20], Plaeiades [19], and Marionette [18]. Other systems that use functional [15] or declarative abstractions [4] must present information from data traces using a different interface. The four hypothetical changes presented illustrate the affects of adding data synchronization to a macroprogram, and are only useful to systems like MacroLab and Hood [12] that make heavy use of data caching. However, the general concept of creating hypothetical changes to illustrate how theoretical changes to a program or execution state *would* affect global state could be applied to aspects of other systems besides data synchronization.

The current implementation of MDB provides *post-mortem* debugging, which means that it collects data about the system at run time and allows the user to inspect program execution after the logs are retrieved. At least some of the underlying concepts, however, could be applied to on-line debugging. For example, the process of logging data instead of events to recreate system state would reduce the amount of data that needs to be collected during on-line debugging, in the same way that it reduces data collection requirements for off-line debugging. Similarly, the logically-synchronous and temporally-synchronous source-level debugging interface could also be used for on-line debugging, and could even be combined with off-line data trace analysis to provide both forward and backward stepping. Hypothetical changes would not be as useful for on-line debugging as they are for off-line debugging, because the user could test changes to the system by simply changing the current state of the system before allowing execution to proceed.

Chapter 4

Dual-Zone: Day/Night Zoning

In order to evaluate the effectiveness of MacroLab as a CPS macroprogramming abstraction and to get a better understanding of the weaknesses of macroprogramming languages for CPS application development a complex sensing and control application is necessary. An occupant-oriented HVAC control system is such an application. This system is developed in three phases, each serving as a case study used to evaluate MacroLab. Case study 1 discusses *Dual-Zone*: a statically zoned system where a house is separated into two zones one of which is conditioned at night while the other is conditioned during the day.

Even though static zoning is a simple approach, controlling such a system based on occupancy could reduce wasted energy that is used to heat and cool unoccupied rooms. However, the energy savings of such a system is not a foregone conclusion due to three key challenges. First, the size of the HVAC system is typically chosen based on the size of the entire house, and so heating or cooling only a fraction of the house with an oversized system for that fraction could prove inefficient. Second, restricting airflow into some rooms will create backpressure in the ducts, which can further reduce the efficiency of the HVAC system by causing leaks in the ducts and at the dampers. Third, most houses do not have insulated interior walls, and the lack of thermal insulation between rooms can lead to heat transfer between the conditioned and unconditioned zones.

This case study demonstrates the feasibility of saving energy by retrofitting a centralized HVAC system to be controlled as a two zoned system where one zone is activated during the day while the other zone is activated at night. A wireless sensor/actuator system that can be cheaply and easily deployed in existing homes is implemented. The system includes 21 temperature sensors and

a wireless thermostat that controls the HVAC hardware and mechanically opens and closes dampers in order to control airflow through the home. The components used cost less than \$500, and a production version is expected to cost considerably less. In contrast, traditional zoning systems often cost more than \$5000. Dual-Zone zoning is deployed in a 7-room, single-story, 1400 square foot house and measured the energy consumption of heating and cooling. The results indicate that the system consumed 20.5% less energy than if the HVAC system were controlled by the existing thermostat over a 20-day experimental period indicating that retrofitting an existing centralized HVAC system for day/night zoning has a potential for substantial energy savings.

4.1 Background and Related Work

Traditional HVAC zoning systems for homes typically separate a house into floors, each of which can be controlled individually. These systems are often installed more for comfort than for energy savings, because a single un-zoned system that operates on multiple floors will often result in a warm top floor and/or a cold bottom floor. Floor-level zoning also makes sense in many homes that have bedrooms on the top floor and living areas on the bottom floor. Floor-level systems have resulted in homeowners saving as much as 20-30% as compared to single zoned systems [99]. However, these systems are expensive and the energy savings can take years or even decades to produce a positive return on investment. Furthermore, it can be difficult to retrofit an existing home with a zoning system.

Numerous studies have explored the effect of zoning a centralized HVAC system, but the results have been mixed and inconclusive. One experiment tested the energy used to heat a single-room with 10 registers and leaky ducts while closing an increasing number of vent registers [100]. The results indicate that closing registers increases the pressure within ducts causing greater duct leakage and reduced system efficiency. However, since all registers were within the same room, this study did not determine whether the reduced efficiency outweighs the savings produced by conditioning a smaller area; all register configurations were conditioning the same sized area.

A subsequent study developed an automated vent louver design for zoning [101], similar to the one developed for day/night zoning and other similar systems [102]. The authors evaluate the system by dividing a house in Danville, CA into four zones and increase the temperature in each zone by 2-5° F. They also increased the temperature in the entire house by the same amount. The results

indicate that it takes less energy to increase the zone temperature per degree than it takes to increase the whole house temperature per degree, since the smaller zones heat up faster than the whole house, allowing the system to turn off sooner. However, this study only measured the transitional time and energy of a zoned system, and it did not measure the steady state energy. In other words, it does not show the difference in energy required to *maintain* a particular temperature in a zone versus the whole house. This distinction is profound, because thermal leakage between adjacent rooms could cause system to also turn back on more quickly, nullifying the energy savings of turning off more quickly. This is often called *short cycling*, and is known to decrease system efficiency as well as reduce the overall lifetime of the equipment.

4.2 Intuition and Preliminary Studies

Before implementing Dual-Zone, two simulation studies were performed to better understand whether such a system should be expected to reduce energy savings, and why. A third analysis of a publicly available smart home data set [103] was carried out to verify that room usage frequencies change throughout the day. These three studies are explained in the following subsections.

4.2.1 Effect of an Oversized HVAC System

In houses with a typical non-zoned central heating and air conditioning system, the size of the system is chosen based on the expected load of the entire house. Therefore, using the same system to heat or cool only a fraction of the house would mean that the system is oversized for the conditioned space. It is well known that oversizing an HVAC system results in reduced efficiency of the system. The first study was designed to determine how much this oversizing would reduce the potential for energy savings of a day/night zoning system.

The EnergyPlus building energy simulation framework [104] is used to simulation the heating of multiple buildings, with increasing size from 5m^2 to 65m^2 . The model buildings had idealized insulation and leakage properties. All buildings were heated with the same sized HVAC system, which was sized for a 65m^2 building. The results are shown in Figure 4.1, which indicate that the amount of energy required to heat a smaller building does indeed decrease, even if the size of the HVAC system remains the same. The sub-linear curve indicates that some efficiency is lost for smaller buildings due to the oversizing of the system. However, this loss in efficiency does not

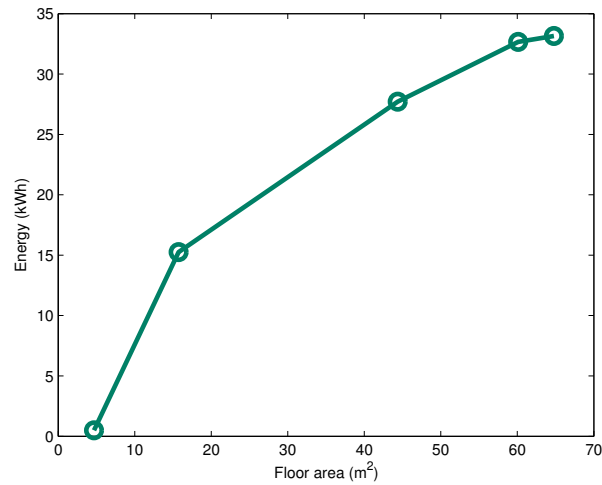


Figure 4.1: With an ideal system, the amount of energy used for conditioning a building is almost proportional to the floorspace being conditioned.

outweigh the gains from heating a smaller space. These results indicate that day/night zoning can be effective, even when applied by retrofitting a home with an existing HVAC system that was sized for the entire house.

4.2.2 Inter-room Leakage

Homes often have thin non-insulated walls and even doors between adjacent rooms, which can reduce the effectiveness of day/night zoning because of thermal leakage between rooms. The second study was designed to explore how much this leakage would reduce the energy savings of a day/night zoning system. The EnergyPlus simulation framework was used to heat a single room in a two-room building. Five variations of the floor plan of the house were used, and the conditioned room had a different number of exterior walls in each variation. The five variations are shown along the x-axis of Figure 4.2, and the energy required to heat the shaded room for each floor plan is shown as a bar graph above each variation. These results indicate that the energy required to condition a room is dramatically reduced as the number of exterior walls of the room decreases. In other words, a neighboring room is a better thermal insulator than an exterior wall, even if the wall between the conditioned room and the neighboring room is not insulated. This result indicates that leakage between conditioned and unconditioned zones will not eliminate the energy-saving potential of day/night zoning.

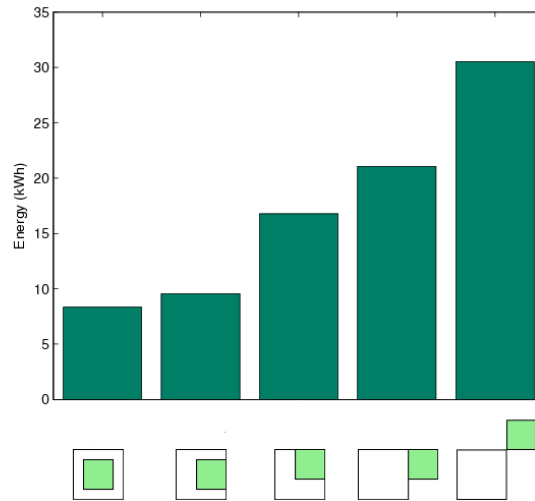


Figure 4.2: The energy required to condition a room decreases as its number of exterior walls is decreased. The x-axis depicts the position of the conditioned room (shaded) with respect to the unconditioned room (unshaded).

4.2.3 Room Occupancy

Finally, the preliminary analysis showed that, even when a home is occupied, the occupants use only a fraction of the house. For example, empirical analysis of one home is shown in Figure 4.3, showing that primarily only one room is used at night, three rooms are used in the evening, and four rooms are used in the morning.

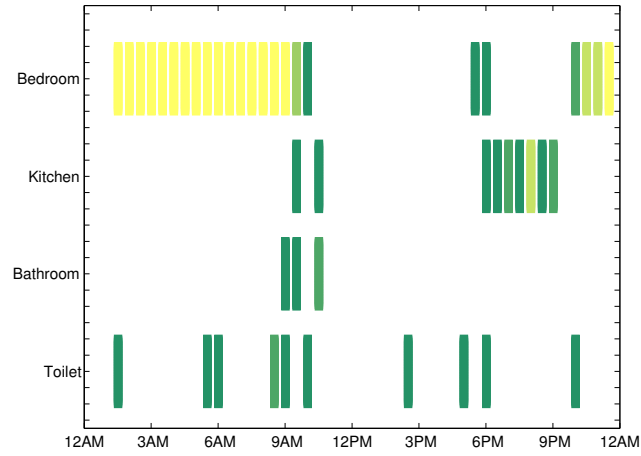


Figure 4.3: The frequency of room usage throughout a day changes. Darker colors indicate lower frequency while brighter colors indicate higher frequency usage with yellow being the highest frequency.

4.3 Implementation

A day/night zoning system was implemented in order to empirically test the ability to save energy with this approach. The implementation involves: (1) sensing temperature at the room-level, (2) controlling air-flow into rooms, and (3) controlling the HVAC system.

4.3.1 Sensing House Temperature

The home's temperature was monitored at a fine granularity by instrumenting the house with wireless temperature sensors placed at various points on the walls. For the deployment discussed in this section, 21 off-the-shelf temperature sensors manufactured by La Crosse Technology [105] were used. Because the temperature across the house is not uniform, one challenge in designing a day/night zoning system is to choose how to process the temperature readings to approximate the true average air temperature in each room. This problem can also be addressed for whole-house conditioning when more than a single temperature sensor is available [106].

Figure 4.4 shows the temporal variations of several temperature sensors placed throughout the house. One sensor is placed in the center of the house, directly in front of the only return register, and therefore is exposed to a mix of air from all rooms. Another sensor is placed on an internal wall of the house, and a third sensor is placed on an external wall. The figure shows that the temperature

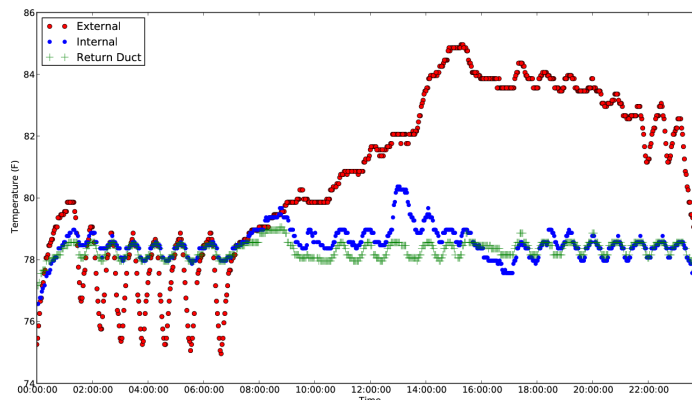


Figure 4.4: The variation of temperature on a sensor placed on an internal wall, an external wall, and near the return duct.

sensor on the internal wall varies with the temperature of the individual room, which is slightly more than the variation of the centrally placed sensor. However, the sensor on the external wall is subject to wild temperature swings. On the left side of the graph, it is clear that the sensor has much greater downward swings than the internal sensors. This is because it is subject to direct air flow from the ducts, which are typically placed on external walls. It is also subject to heat that concentrates around the window mid-day. Because of these large temperature fluctuations, only sensors on the internal walls of each room were used: the temperature in a zone was calculated as the average of the temperatures of each of the internal sensors in the rooms comprising the zone.

4.3.2 Controlling Air-flow into Rooms

In order to control the airflow into individual rooms *active registers and dampers* that can be wirelessly opened or closed were designed and built. While controllable registers are commercially available, they actuate based on either preset temperatures or temporal schedules. Commercial active registers that are controllable through a remote control would be hard to integrate with the day/night zoning's wireless control system and such registers are expensive, costing over \$50 each. By retrofitting passive registers with servo motors, active registers were obtained for under \$20 each, excluding the cost of the radio and microcontroller. These registers integrated wirelessly with the rest of the Dual-Zone infrastructure. The active register design improved through three generations as shown in Figure 4.5. The registers were implemented using commercial, off-the-shelf (COTS) components including an operable register, a servo motor, and a small amount of custom

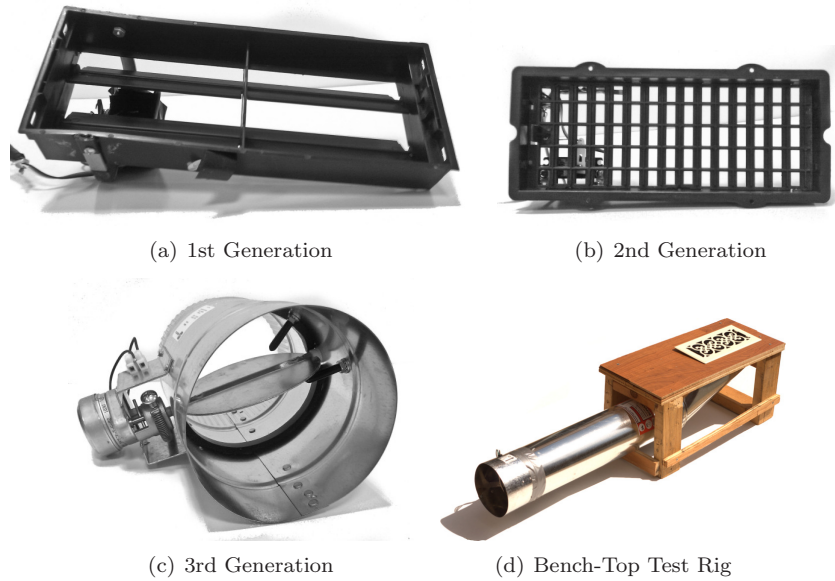


Figure 4.5: (a) Version 1 uses servo motors and a rotating louver design, but exhibited too much leakage. (b) Version 2 uses a sliding gate design to solve the leakage problems, but causes too much noise. (c) Version 3 is a commercial in-line damper with a servo motor used for traditional zoning applications. (d) The bench-top test rig used to verify that the second generation wirelessly controlled active registers have almost no air leakage.

circuitry (Figure 4.6). These components resulted in a cost of less than \$20 per register excluding the cost of the TelosB mote, which was used for wireless communication. These registers were inspired by several prototypes and even commercial versions of similar hardware that are currently available [100, 101, 107, 108], but go beyond these devices by integrating them into a cooperative, wireless system.

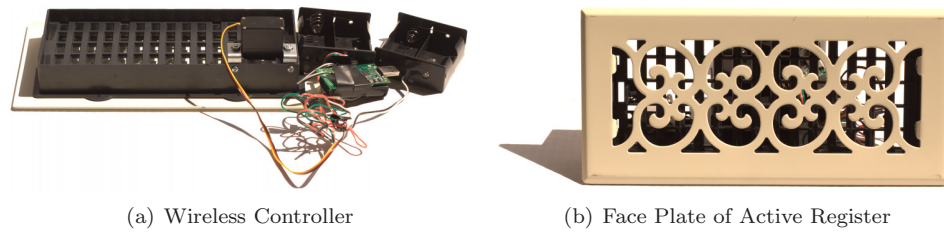


Figure 4.6: A wirelessly controlled active register built by augmenting a standard vent register with a TelosB mote and a servo motor.

The effectiveness of the registers were measured using the bench-top testing framework shown in Figure 4.5(d). The first generation registers were not very efficient at blocking air when closed. The second generation registers improved this aspect by blocking nearly 100% of airflow but was noisy

when opening and closing. Due to these reasons dampers used in commercially implemented zoned systems were used as the third generation of airflow controllers.

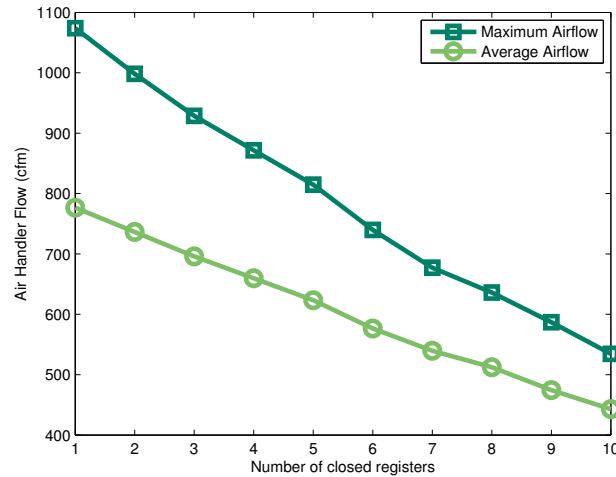


Figure 4.7: As more registers are closed, some efficiency is lost and the total total air volume output by the system decreases.

The effectiveness of these registers at directing airflow into different rooms was measured using a Kestrel 4100 Pocket Air Flow Tracker manufactured by Nielsen-Kellerman [109]. This sensor is placed above the register and provides a measure of airflow in terms of cubic feet per minute (CFM) that is coming out of the register. This measurement is based on the known size of the register and the speed of the air. Figure 4.7 shows that the total airflow coming from all registers is reduced as an increasing number of registers are closed. When all registers are open, the average airflow is approximately 800 CFM, which matches the specification of the air handler in this house. However, as more registers are closed, the average airflow approaches 450 CFM, which is almost half. Total airflow does not approach zero because some air escapes even from the closed registers. This result verifies that closing registers does decrease the overall efficiency of the system because it reduces the total airflow output, as suggested by [100]. Therefore, actively cooling only half the house would not cause double the amount of air to be available to the cooled zone, because some air is lost due to backpressure, increased duct friction, duct leakage, and leakage from the closed registers.

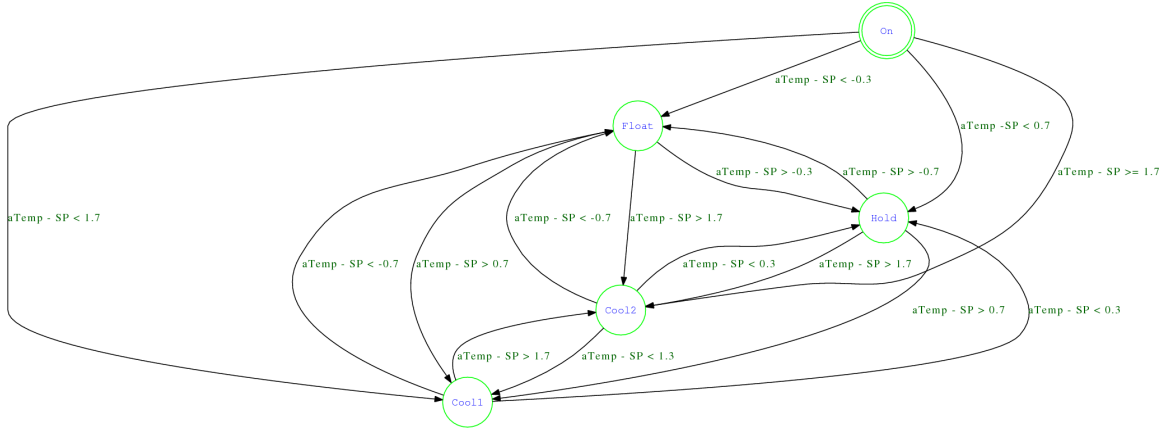


Figure 4.8: The zoning controller attempts to maintain the average temperature of the active zone ($aTemp$) at the desired setpoint (SP) by transitioning the system between five states.

4.3.3 Controlling the HVAC System

Dual-Zone uses a simple state machine (Figure 4.8) to control the HVAC system through four possible stages: *Float*, *Hold*, *Cool 1*, and *Cool 2*. Cool 1 and Cool 2 are intended to represent different stages of the HVAC system in which the compressor and hair handler operate at different cooling capacities. Hold causes the HVAC system to maintain the current temperature at the thermostat, and Float causes the HVAC system to turn off.

| State | Action |
|-------|---------------------------|
| Float | $ThermSP = ThermTemp + 1$ |
| Hold | $ThermSP = ThermTemp$ |
| Cool1 | $ThermSP = ThermTemp - 1$ |
| Cool2 | $ThermSP = ThermTemp - 2$ |

Table 4.1: The operating stage of the HVAC equipment was controlled by adjusting the thermostatic setpoint $ThermSP$ with respect to the temperature that was sensed by the thermostat $ThermTemp$.

In order to control the HVAC equipment, the Dual-Zone controller must interface through an Internet-controllable thermostat manufactured by BAYweb [110]. However, the BAYweb thermostat only allows its setpoint to be changed; it does not allow direct control over the equipment. In order to control the equipment, therefore, the setpoint of the thermostat $ThermSP$ was modified to be higher, lower, or equal to the temperature measured at the thermostat $ThermTemp$. When the equipment needed to be put into the *float* state, a setpoint that is higher than the current temperature was used. This causes the thermostat to turn off the equipment. Similarly, when it was necessary to hold or lower the temperature, a setpoint that is the same as or lower than the current temperature,

respectively, was used. To lower the temperature quickly, i.e. to use stage *Cool 2*, a setpoint that is two degrees lower than the setpoint was used. This exploits the PI controller that is built into the thermostat, which causes the equipment to go into high stage cooling when the temperature is two degrees from the setpoint for more than 5 minutes. The operation of the system is summarized in Table 4.1. This coarse-grained control over the equipment is not ideal and could have caused some loss of efficiency and energy waste. In case study 2.5 a custom thermostat is built to provide finer grained control that produced better results

4.3.4 Software Implementation

Dual-Zone was deployed in an 8-room, single story, 1200 square foot residential building shown in Figure 4.9. For simplicity, the house was divided into two zones. The red zone composed of the living room, dining room, and kitchen is actively conditioned between 8:00 AM and 9:30 PM while the blue zone composed of the bedroom, nursery, toilet, and mudroom is conditioned between midnight and 8:00 AM. Between 9:30 PM and midnight the whole house is conditioned. This approach is compared to conditioning of the whole house using an off-the shelf programmable thermostat manufactured by *BAYweb* [110]. In both cases, the setpoint temperature of the house is controlled by the occupants. This means that the experiments measure the energy required to keep the occupants comfortable with both systems, as opposed to keeping the space at a particular setpoint.

The software that controlled the system was written using Python. Due to the fact that the MacroLab function libraries and hardware drivers are very immature, and it would have taken a considerable amount of time and effort to incorporate the required function into the library, the actual implementation was done only in Python. The programs written in Python read from the temperature and motion sensors and sent actuation commands to active registers and the HVAC hardware. Figure 4.8 presents a state machine of the controller that was implemented in Python. The controller transitions the HVAC system between the four states listed in Table 4.1 depending on the current state, the average temperature ($aTemp$) in the active zone and the setpoint (SP) using hysteresis to prevent rapid fluctuations between states. This program was executed to collect the data used to analyze Dual-Zone.

As a case study, the Dual-Zone controller software was re-implemented using MacroLab as shown in the code in Figure 4.10. The MacroLab implementation is based on the temperature and motion sensor reading being abstracted as macrovectors that can be indexed using room IDs. Actual usage of

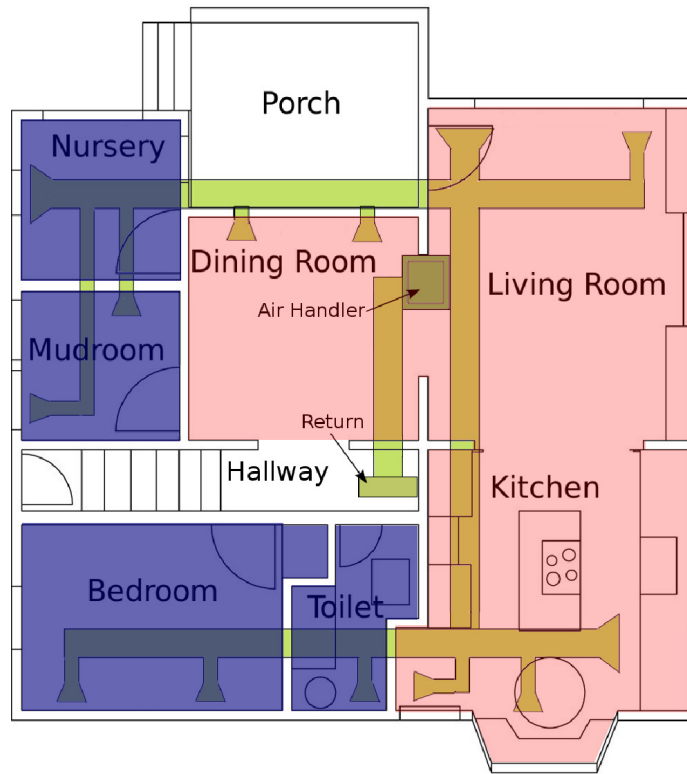


Figure 4.9: The residence in which experiments were carried out with green ducts terminating in registers that can be opened or closed. The rooms that compose the day zone are shaded in light red and the rooms that compose the night zone are shaded in dark blue. The red circles show locations of some of the temperature sensors used for HVAC control.

this program to control Dual-Zone would require the implementation of two driver functions: *dbRead* to read from a database and *hvacActuate* to actuate the HVAC system and active registers. This code could execute with the COTS sensors used to implement Dual-Zone by providing implementations of *weatherDirect* and *X10* that read sensor values from the Weather Direct web interface and X10 base station or query the appropriate database tables. If other sensors, such as SnapPys or motes, that could be programmed were used for sensing instead, sensor driver implementations that push, or aggregate, data could be used to optimize the execution of this code by just replacing the *type* parameter in the *getMotes* function as described in Section 4.4.2.

```

1 RTS = RunTimeSystem();
2 weatherdirect = RTS.getMotes('type', 'weatherdirect');
3 tempSensors = SensorVector(weatherdirect, 'temperature');
4 x10 = RTS.getMotes('type', 'X10');
5 motionSensors = SensorVector(x10, 'motion');
6 zones = uint8([3 4 5], [1 2 6 7]); % Day/Night zones
7 motionSensorIDs = uint8([8 1 9], [2 6], [10 5], [4], [7 11], [12 14], [3 15]);
8 tempSensorIDs = uint8([1 3], [2], [6 7], [4 9 11], [12 13], [5 14], [8 10]);
9 nightStart = [0 0 0 2 0 0];
10 nightEnd = [0 0 0 7 0 0];
11 curState = 'On'
12 every(60000)
13     mode = dbRead('zoning', 'mode', 'latest')
14     motionVals = motionSensors.sense();
15     tempVals = tempSensors.sense();
16     curTime = clock;
17     curNightStart = nightStart;
18     curNightStart(1:3) = curNightStart(1:3) + curTime(1:3);
19     curNightEnd = nightEnd;
20     curNightEnd(1:3) = curNightEnd(1:3) + curTime(1:3);
21     curZone = {};
22     if datenum(curTime) <= datenum(curNightStart) && datenum(curTime) > datenum(
        curNightEnd)
23         curZone = zones{1};
24     else
25         curZone = zones{2};
26     end
27     curZoneMotion = [];
28     curZoneTemps = [];
29     for room = curZone
30         curZoneMotion = [curZoneMotion motionVals(motionSensorIDs{room})];
31         curZoneTemps = [curZoneTemps tempVals(tempSensorIDs{room})];
32     end
33     if sum(curZoneMotion) > length(curZoneMotion) / 2
34         SP = dbRead('zoning', 'setpoint', 'latest');
35     else
36         SP = dbRead('zoning', 'setback', 'latest');
37     aTemp = mean(curZoneTemps);
38     if mode == 'heat'
39         deltaTemp = SP - aTemp;
40     else
41         deltaTemp = aTemp - SP;
42     end
43     curState = getNextState(curState, deltaTemp);
44
45     hvacActuate(mode, curState, curZone);
46 end

```

Figure 4.10: MacroLab implementation of Dual-Zone.

```

1 function getNextState(curState, deltaTemp)
2     if curState == '0n'
3         if deltaTemp < -0.3
4             curState = '-1'; % float
5         elif deltaTemp < 0.7
6             curState = '0'; % hold
7         elif deltaTemp < 1.7
8             curState = '1'; % heat/cool 1
9         elif deltaTemp >= 1.7
10            curState = 2; % heat/cool 2
11    elif state == '-1'
12        if deltaTemp > -0.3
13            curState = '0';
14        elif deltaTemp > 0.7
15            curState = '1';
16        elif deltaTemp > 1.7
17            curState = '2';
18    elif state == '1'
19        if deltaTemp < -0.7
20            curState = '-1';
21        elif deltaTemp < 0.3
22            curState = '0';
23        elif deltaTemp > 1.7
24            curState = '2';
25 end

```

Figure 4.11: MacroLab function to calculate transition in hysteresis state machine.

4.4 Evaluation

To evaluate Dual-Zone the control of the HVAC system is alternated between single-zoned whole house control and day/night zoned control over a twenty day period, such that each system ran every other day. This experimental procedure was selected to minimize the effect of changing weather patterns on energy statistics. Both systems executed for a total of 10 days. The energy consumed by all systems in the house was monitored using The Energy Detective (TED) [111] real-time in-home energy management system and the amount of energy used by the HVAC system was deduced using the operation logs generated by the BAYweb thermostat.

4.4.1 Zoning Evaluation

Figure 4.12 shows the energy consumed in conditioning a house using Dual-Zone and whole house conditioning. This graph indicates that whole-house conditioning consumed 20.5% more energy than the prototype implementation of Dual-Zone, on average.

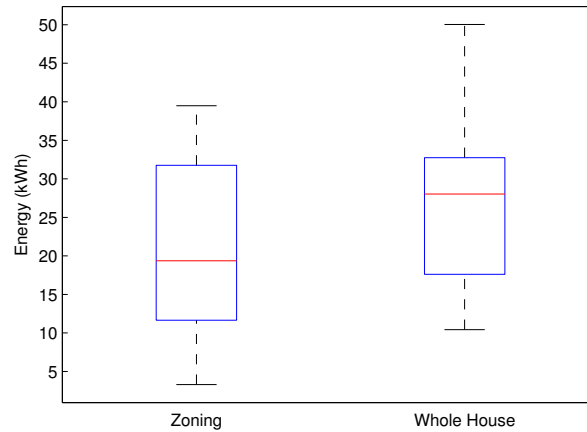


Figure 4.12: The implementation of Dual-Zone uses 20.5% less energy than whole house cooling on average.

Figure 4.13 shows the actual energy consumption for each day as a scatter plot, with the average temperature of for that day on the x-axis, the energy consumed on the y-axis, and the control algorithm shown as the color of the scatter point.

Figure 4.14 shows how the temperature of several rooms in different zones vary as the temper-

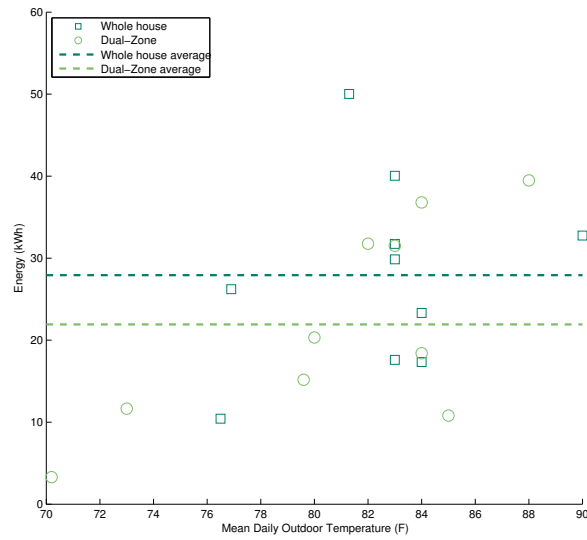


Figure 4.13: The dotted lines indicate the average energy used over the experimental period.

ature in the active zone was dropped from 76 to 72 degrees. In this graph, the bottom three lines shown the temperature of conditioned rooms over time while the top two lines show the temperature of unconditioned rooms. Although some leakage is evident, particularly into the top line, the temperatures of the unconditioned rooms remains substantially higher than the conditioned rooms. These temperature traces explains how Dual-Zone is able to save energy by reducing the size of the space that must be conditioned.

In order to better understand these results, Figure 4.15 illustrates how effective the active registers were at activating and de-activating the red and blue zones. It is clear from this figure that the greatest airflow in a zone is obtained when the registers in the other zone are closed. However, air flow to the inactive zone does not stop, nor does it all get directed to the active zone. In future work, the active register system needs to be improved to produce better energy saving and thermal insulation results.

4.4.2 Macroprogramming Discussion

For case study 1, MacroLab proved to be a much more convenient language in which to program than Python. The Dual-Zone controller presented in Figures 4.10 and 4.11 is written in fewer than 100 lines of MacroLab code. Making a conservative estimate of 200 lines of code with the addition

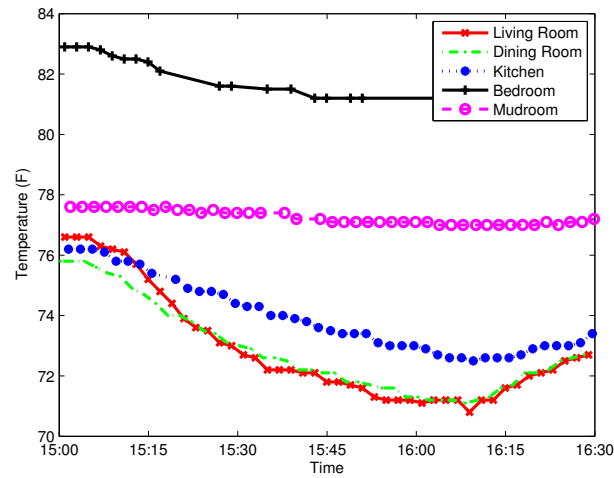


Figure 4.14: Temperature response of both conditioned and unconditioned rooms as the active zone temperature is dropped from 76 to 72.

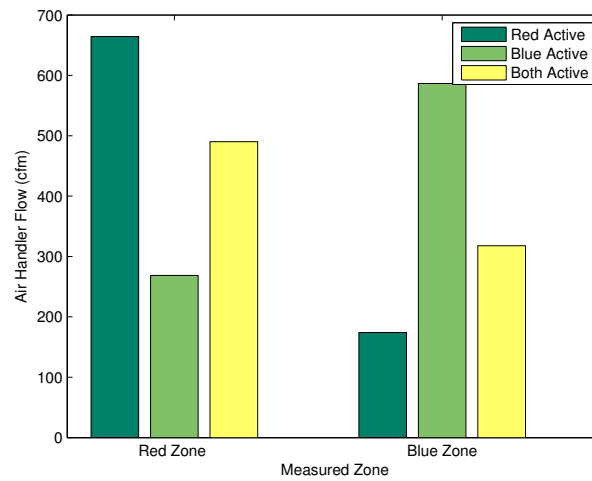


Figure 4.15: Activating different zones does not redirect all air flow from one zone to another, but does affect air flow substantially.

of low-level functionality, such as hardware drivers, implementing Dual-Zone in MacroLab is much more efficient in terms of code size compared to the over 500 lines of code written for the Python implementation. An added advantage of MacroLab is the ability for the run-time system (RTS) to automatically optimize the code based on available hardware or the topology of the network. The following subsections describe some of the optimizations that would have been possible with a MacroLab implementation of Dual-Zone.

In-Network Data Processing

In the implementation of Dual-Zone using Python, all temperature sensors transmitted readings periodically. By providing the MacroLab RTS with *tempsensor.sense* and *motionsensor.sense* functions that are aware of a sensor's location and the current time, node-level code could be generated where sensors only transmit their temperature or motion readings if they are in the currently active zone. In essence, the following portion of the macroprogram is pushed into the network, to be executed at the sensors rather than the base station.

```

1 if datenum(curTime) <= datenum(curNightStart) && datenum(curTime) > datenum(
    curNightEnd)
2   curZone = zones{1}
3 else
4   curZone = zones{2};
5 end
6 curZoneMotion = []
7 curZoneTemps = []
8 for room = curZone
9   curZoneMotion = [curZoneMotion motionVals(motionSensorIDs{room})]
10  curZoneTemps = [curZoneTemps tempVals(tempSensorIDs{room})]
11 end

```

This simple optimization could almost halve the number of messages sent by a sensor, potentially doubling their battery life. For temperature sensors which have an average lifetime of a year, an increase to two years is very beneficial.

In-Network Aggregation

Another optimization that MacroLab could enable is in-network aggregation. Each of the two zones could form independent neighborhoods with an aggregation tree built from the furthest sensors towards the base station. The average temperature of the zone could be aggregated by each sensor calculating the sum of its and its children's temperatures and passing it up the tree along with a count of the number of temperatures that were summed. The base station would receive a temperature sum from the zone along with the total number of sensors within the zone, which can be used to calculate the average temperature of the zone. Without in-network aggregation, for a set of N

sensors where sensor i is at a depth of d_i from the base station, the number of messages transmitted each time is $\sum_{i \in N} d_i$. Using in-network aggregation reduces the number of messages sent to equal the number of sensors in the network, N . In-network aggregation only proves beneficial if the system involves a multi-hop network since in a single-hop network all transmissions can be received at the base station where computation could be carried out centrally.

Data Fusion

Finally, the MacroLab RTS could use data fusion to minimize the number of messages passed to the base station by aggregating the motion sensor and temperature sensor values at a leader node within each zone. The leader node would execute the following portion of the macroprogram:

```

1 if sum(curZoneMotion) > length(curZoneMotion) / 2
2   SP = dbRead('zoning', 'setpoint', 'latest');
3 else
4   SP = dbRead('zoning', 'setback', 'latest');
5 aTemp = mean(curZoneTemps);
6 if mode == 'heat'
7   deltaTemp = SP - aTemp;
8 else
9   deltaTemp = aTemp - SP;
10 end

```

By processing both the temperature and motion sensor data within the network, the number of messages passed by both sensors up to the base station can be minimized. This approach provides savings only for very large networks where all the nodes in a particular zone are multiple hops from the base station. The RTS can optimize this further by keeping track of the previous *deltaTemp* value and only transmitting a *deltaTemp* value if the new value exceeds the previous value by a threshold.

4.5 Conclusions

This study demonstrates how MacroLab can be used to simplify the software implementation of a static day/night zoning system. It also shows the power of deployment-specific code decomposition

(DSCD) in optimizing an application to suit network topologies and available hardware. The case study involves minimizing the energy consumed for heating and cooling homes by conditioning only occupied spaces. The implementation of Dual-Zone shows that a centralized HVAC system can be cheaply and easily retrofitted to save energy. Even with leakage from the registers and imperfect isolation between rooms, whole house zoning consumed 20.5% more energy than this system over the course of a 20-day study. While longer-term studies are necessary to eliminate potential effects of weather during the study, these results are promising and warrant further investigation into this approach, especially in light of the shortcomings of the prototype, as described in Section 4.3, and the fact that only multi-room zones were evaluated instead of room-level zones. Case study 2.5 builds on Dual-Zone to implement a room-level zoning system which is used to show how the evolution of software is simplified by MacroLab and the limitations of macroprogramming abstractions as system complexity increases.

Chapter 5

RoomZoner: Room-Level Zoning

Case study 1 (Chapter 4) was a preliminary analysis into the feasibility of retrofitting a centralized HVAC system to enable two distinct zones: a day zone and a night zone. Case study 2 involves the implementation of *RoomZoner*: a room-level HVAC zoning system where each room is individually conditioned based on its occupancy and temperature. RoomZoner dynamically change zones in response to occupancy and temperature changes within rooms. An occupancy assessment technique that relies on simple motion sensors is used to decide when rooms become occupied and vacant and a simple thermal model of the house to predict the effect of control decisions. Using this information RoomZoner responds to changes in occupancy by redirecting conditioned air through the house. This approach is based on the concept of Micro-zoning which allows an HVAC schedule for individual rooms to be customized [112]. Rose and Dozier observed that micro-zoned systems report energy savings of up to 43% compared to conventional HVAC systems with larger zones [113]. The main contribution of RoomZoner is that each zone is controlled based on occupancy.

RoomZoner involves a novel occupancy assessment technique and a zone control algorithm that ensures the safety of HVAC hardware while attempting to minimize energy wastage without compromising occupant comfort. The occupancy assessment technique enables simple motion sensors, like those found in most residential security systems, to be used to accurately infer rooms that are occupied and distinguish them from rooms that being used only temporarily such as those that serve as hallways between other rooms. The zone controller attempts to maintain occupied rooms at a comfortable setpoint and ensure unoccupied rooms can be conditioned to the setpoint within a short time of occupancy. It also ensures the safety of the HVAC hardware by ensuring the system is not

being turned on and off too frequently or too much pressure is being built up within the ducts due to the dampers to too many rooms being closed. A study carried out over two months indicates that room-level zoning, with RoomZoner, consumes 14% less energy than whole house conditioning.

5.1 Background and Related Work

Several previous studies have explored the possibility of room-level zoning, but the conclusions of these studies have been mixed and inconclusive [99–101]. RoomZoner addresses this shortcoming with a long-term deployment in a residence which is instrumented to enable occupancy to be monitored and the HVAC system controlled in response to the occupancy. RoomZoner responds to changes in occupancy using an occupancy assessment technique that relies on simple occupancy sensors. Many different techniques for tracking occupants through a home, identifying them, and understanding what activities they are doing have been proposed. Until recently, however, existing technologies have all been too expensive or intrusive for use in energy conservation applications. For example, some systems require the user to wear a tag [114], or to actively trigger a biometric sensor such as a thumbprint or retina scanner. Other systems require cameras to be installed in the home [115] and identify people, locations, and activities using gait analysis, form matching, or face recognition. However, cameras are often perceived to be invasive to personal privacy. Other systems require structural changes to the home such as the installation of smart floors, which incur a high initial cost and effort that cannot be justified by applications in energy conservation. Finally, some systems detect user activities by requiring a large set of training data that must be laboriously collected for weeks or even months after the original sensor deployment [116].

Unlike other smart home applications that rely on fine-grained activity recognition, RoomZoner does not require cameras or wearable tags that may be considered intrusive to the user; in contrast to other smart home applications such as medical monitoring and security, this domain can tolerate a small loss in accuracy in favor of cost and ease of use. Therefore, RoomZoner utilizes commercial off the shelf (COTS) motion sensors, which cost approximately \$5 each. These sensors are wireless and can be installed in minutes using double-sided tape. Evaluations of a similar system in eight homes have been able to identify occupancy and sleep activities with over 90% accuracy [117, 118]. We evaluate the raw sensor readings with a novel occupancy assessment approach to distinguish rooms that are actively being used from those that are transiently occupied as residents pass through them.

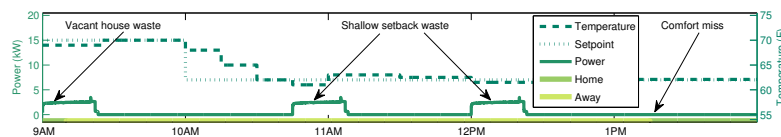


Figure 5.1: Programmable thermostats cause substantial energy waste and discomfort.

RoomZoner uses a reactive thermostatic control scheme. Reactive control is one of the three common types of HVAC thermostatic control mechanisms, the other two being manual and programmable thermostats [119]. While thermostats differ in the way they operate depending on what category they fall into, they all attempt to trade off energy savings for occupant comfort.

Manual thermostats maintain the temperature of the house, or zone, it is monitoring at a temperature to which it is currently configured, the setpoint. With conscientious users, who setback the temperature when they leave the house and go to bed and return the temperature to a comfortable level only when they are active around the house, manual thermostats can be the most energy efficient type of thermostat. Yet, they place a tremendous burden on the user to constantly set the temperature based on his/her current activity. Also, the fact that manual thermostats only start heating or cooling a house after the residents return, or wake up, results in the resident having to endure periods of discomfort while the house is warming up or cooling down. These reasons result in over 65% of residents with manual thermostats not switching to a setback temperature when they vacate their houses [120].

Programmable thermostats operate on a pre-defined *setback schedule*: the house is conditioned to a setpoint temperature when the occupants are typically active, and floats to a more energy efficient setback temperature when the occupants are typically away or asleep. This approach wastes energy in several ways, as illustrated by the composite of data traces in Figure 5.1 that were collected from a home using a programmable thermostat. First, the occupants leave the home shortly after 9am, but the system wastes energy because it is scheduled to continue heating the home until 10am (left side). Second, the setback temperature is well above safety limit for a home, causing energy consumption even while the house is vacant (center). This type of *shallow setback* is typically used as a safety precaution, in case the building actually is occupied at that time. Third, the occupants become uncomfortable when they return shortly after 1 pm because the system is not scheduled to heat the house until much later. This causes people to reduce their use of setback schedules, or stop using them altogether. Over 50% of households that have programmable thermostats are reported

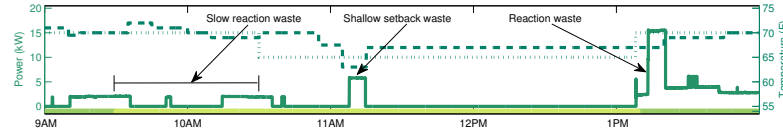


Figure 5.2: Occupant-oriented thermostats have three sources of energy waste.

to not use setback periods at night or during the day [121]. In contrast, households with the simpler dial-type thermostats can easily adjust temperature settings before going to sleep or leaving the house, and as a result actually save more energy on average than households with programmable thermostats [121, 122].

Reactive thermostats use motion sensors, door sensors, or card key access systems to turn the HVAC equipment on and off based on occupancy. However, preliminary studies of such systems in residential buildings have demonstrated less energy savings than programmable thermostat and even increased energy usage by up to 10% [123]. Some commercial buildings are beginning to use occupancy sensors to create a tighter link between occupancy and HVAC control with reactive thermostats that use motion, window, and/or door sensors to turn the HVAC system on or off after detecting that the occupants have left or returned to a space. For example, the Telkonet SmartEnergy [124] control system allows the user to define a maximum recovery time parameter, which is set by the user. When the space is unoccupied, the system maintains a setback temperature from which it can return to the setpoint within the specified recovery time, once the occupants return. The system estimates the response time based on building and system parameters as well as current weather conditions. Other similar commercial systems include the Verdant [125], Viconics [126], and PECO [127] systems. While reactive thermostats are a step in the right direction, they are almost always applied to hotel rooms because they rely on the simplicity of hotel rooms and the keyed entrance to identify occupants; more complex spaces with multiple rooms, entrances, and occupants require more advanced sensing technology, such as the techniques discussed in this proposal.

Thermostatic control wastes energy due to occupancy patterns varying from the pre-defined schedule, and the system having to maintain a setback temperature close to the setpoint temperature so as to minimize discomfort if the occupants become active unexpectedly (Figure 5.2). In order to remove the burden of having the program a thermostat based on occupancy patterns, the *self-programming thermostat* which automatically chooses the optimal setback schedule was proposed [123]. This system does not fully solve the problems because occupancy patterns change every

day, and so *any* static schedule must either waste energy or sacrifice occupant comfort.

Smart Thermostat was implemented as an extension to the self-programming thermostat. It incorporated dynamic actuation in order to minimize the energy wastage and loss of comfort due to variations in occupancy patterns [128]. A Hidden Markov Model (HMM) was trained with data collected from eight homes over a period of two weeks using leave-one-out cross-validation. This model was used to predict occupancy and control a mult-stage HVAC system. While this approach goes a long way in minimizing the energy consumed by not conditioning an unoccupied house, there is still a lot of savings to be had by not conditioning unoccupied spaces of an occupied house.

Gupta et al. [129] add GPS-control to traditional thermostats by using information from location aware mobile phones to augment thermostats with the ability to control HVAC systems using travel time. By estimating how long it takes for the residents to return to an unoccupied home, the thermostat can be placed in a “just-in-time” travel-to-home-time mode so that it starts conditioning the house in time for the residents’ arrival, thus staying in a lower-power setback mode for longer. Through simulations, the authors demonstrate energy savings of up-to 7% in certain households. While this system is one solution to the problem of efficiently using programmable thermostats, it generally results in lower savings than programmable or manual thermostats and does not provide any energy savings in heating or cooling occupied houses.

Mozer et al. present the *Neurothermostat* which uses daily occupancy schedules and a neural network trained on five months of occupancy data in order to control an HVAC system [130]. The authors demonstrate that the Neurothermostat results in a lower unified cost, defined as a combination of energy usage and occupant discomfort.

Previous researchers have also investigated using multiple temperature sensors in a building with a single thermostat [106]. This work, done in simulation, demonstrated an increase in user comfort by using targeted comfort control strategies. Others retrofitted a house in Danville, CA with wirelessly controllable registers and demonstrated that energy savings is possible by directed air into localized zones [101]. This work did not use occupant information to control the HVAC system.

Commercial buildings often use zoning systems that divide a single floor into multiple rooms. This is especially common in hotels, banquet halls, and office buildings. For example, the discharge-air-regulation technique (DART) uses temperature sensors to control the HVAC fan speed [131]. Other systems include the Millennial Net [132] and Siemens APOGEE [133]. Just like the residential zoning systems, these solutions are expensive and are much easier to add to a new installation.

Similarly, micro-environment systems (also called task-ambient conditioning) allow a worker in an office building to have fine-grained control over the ambient conditions around his or her working space, typically a desk. Several systems, including Personal Environments from Johnson Controls [134] and Habistat from Interface Architectural Resources, are currently commercially available. The individually controlled spaces are not insulated from each other and operate within a single thermal zone. These systems are designed for occupant comfort over energy efficiency. The systems can produce some energy savings by not conditioning desks that are not occupied, and several studies have shown substantial savings of micro-environment systems [112, 113, 135]. However, the cost of these systems is between \$20,000 and \$100,000 per desk, which is too large to produce a positive return on investment. Furthermore, this approach is designed for offices and would be difficult to transfer to homes, where usable space can be more difficult to instrument than a desk or cubicle.

Finally, there have been patents filed for occupancy-based zoning of HVAC systems using security systems [136] or motion sensors [137] to detect occupancy. While these systems attempt to solve the problem addressed by RoomZoner, the effectiveness of their approach is not evaluated. Also, these systems fail to address hardware safety concerns that arise with implementing room-level zoning using a centralized HVAC system. RoomZoner is cognizant of the short-cycling and back-pressure that could reduce the lifespan of HVAC hardware and attempts to minimize the potential damage to hardware.

5.2 Occupancy Assessment

RoomZoner aims to minimize the energy wasted when rooms that are unoccupied are conditioned. Therefore, the goal of occupancy assessment is to detect when rooms are used. In an ideal scenario, at night when two residents are asleep, only the bedroom would be detected as being occupied. When one of the residents wakes up in the morning and goes into the bathroom, the bathroom would be detected as being occupied in addition to the bedroom. When the resident leaves the bathroom and enters the kitchen, the bathroom would be detected as being unoccupied and the kitchen would be detected as being occupied. Thus, as soon as a room starts being used it should be detected as occupied and as soon as it stops being used it should be detected as unoccupied.

Such room usage detection is ideal for the control of systems such as lights or faucets where the required service is provided almost immediately when it is activated and can be deactivated instantly

when the service is not needed, with no repercussions. An HVAC system, in contrast, takes time to heat or cool a space and turn off, and therefore cannot react to occupants entering and leaving rooms without sacrificing comfort and efficiency. Therefore, assessing room occupancy goes beyond merely detecting when a room starts being used and stops being used. The occupancy assessment algorithm for RoomZoner has to find room usage time constants commensurate with equipment operation periods. For instance, if the HVAC system takes 15 minutes to heat the bathroom by one degree and the resident enters and leaves the bathroom within a minute, detecting this occupancy and reacting to it would waste energy while providing no benefit in terms of comfort. Therefore, in assessing room occupancy, the system has to take into consideration the HVAC equipment operation parameters.

Another issue that has to be considered when assessing occupancy is the short cycling of HVAC equipment. Short cycling is the transition of the HVAC system from an operating stage to a lower stage before the manufacturer recommended minimum time. For instance, changing from stage 2 heat to stage 1 heat or turning off from stage 1 cool before the minimum operating time for that stage. Short cycling the HVAC system can cause equipment damage due to increased wear and tear as a result of frequent equipment cycling and not allowing the pressure within the system to equalize between cycles.

In addition to potentially damaging hardware, short cycling could also waste energy and subject the residents to discomfort. For instance, if a resident spends most of his/her day in the living room but goes to the kitchen periodically to get water or a snack, if the system did not consider short cycling, the HVAC would turn off whenever the resident left the kitchen assuming the living room was at the setpoint and the kitchen drifts away from the setpoint due to being used infrequently. This turning on and off of the equipment would waste energy while not providing any benefit to the resident. Also, by allowing the kitchen to drift away from the setpoint, the system would expend a large amount of energy in conditioning it when it is used for a long period of time, such as when the resident is preparing a meal. Similarly, short cycling the system would mean a room that is used infrequently in conjunction with another room would never reach the setpoint since the system turns off whenever the room stops being used. Thus, every time the occupant enters the room s/he would be in discomfort. If occupancy assessment only considered if a room was used or not in order to define it as occupied or unoccupied, and actuated the HVAC system in response to this information, short cycling would be inevitable. Thus, a more sophisticated approach to occupancy assessment is

needed as will be described below.

5.2.1 Challenges

The challenges in assessing occupancy in most houses arise from room usage durations not matching equipment operation periods and errors inherent in sensing hardware. The mismatch in room usage durations and equipment operation periods can manifest itself in *passageway rooms*, *multi-room usage*, and *short-term room usage*. Inadequacies of hardware usually manifest themselves as false positives or negatives. In this section these four challenges to occupancy assessment are described.

Passageway Rooms

Passageway rooms are rooms, such as hallways, which connect two or more other rooms. These could be central rooms in a house that are surrounded by other rooms. Passageway rooms pose a challenge because they are constantly in use throughout the day, yet for very short durations of time. Therefore, trading off between saving energy and ensuring resident comfort becomes difficult. If comfort was of utmost important passageway rooms would be considered as occupied for most of the day so that whenever a resident passes through them they would be at the comfortable setpoint temperature. Yet, because the amount of time a resident spends in this rooms is a small fraction of a complete day, most of the energy expended in maintaining the room at the setpoint would be wasted. Thus, a successful occupancy assessment algorithm has to identify a room as a passageway room so as not to waste energy conditioning when it is not occupied, yet attempt to maintain it at close to the setpoint so that when occupants pass through the room, as they are likely to do frequently throughout the day, they are not discomforted.

Short-term Room Usage

In addition to passageway rooms, there are rooms that are usually not used for long periods of time. For instance storage rooms, bathrooms, kitchens, and wardrobes are entered for short periods of time throughout the day. These rooms cannot be conditioned in response to usage because, in most instances, the rooms are not occupied for a sufficient period of time for conditioning to be effective. Differentiating short-term room usage from regular usage is a challenge that has to be addressed in order to prevent the wastage of energy.

Multi-Room Usage

There are many instances when rooms are used in groups. For instance a resident may watch the news on television while preparing dinner, so that s/he alternates between the kitchen and the living room, or set the dining table while keeping an eye on a dish on the stove, alternating between the dining room and kitchen. Such usage patterns cannot be identified and exploited if the occupancy assessment algorithm defines room usage based on a resident entering and leaving a room. In other words, a room does not necessarily start being used the moment a person enters it and stop being used the moment the person leaves. The occupancy assessment algorithm has to detect multiple rooms being used concurrently so that they maybe conditioned as a group, increasing the efficiency with which the HVAC system operates.

False Positives/Negatives

False positives and negatives are common errors associated with simple binary occupancy sensors. PIR and X10 sensors can be triggered by shadows or the changes in light levels due to the movement of the sun for instance resulting in false positive sensor readings. These sensors also have a limited sensing radius through which people have to move to be detected. Moving through blind spots within a room can result in false negatives sensor readings. False positives have to be overcome by filtering out spurious sensor firings while false negatives have to be minimized by increasing the number of sensors in a room so as to reduce the blind spots. Yet, these inadequacies of binary sensors have to be considered when assessing occupancy.

5.2.2 Approach

The implementation of occupancy assessment for RoomZoner involves learning room usage patterns using historical data and constantly evaluating the sensor firings to identify the occupancy patterns exhibited by occupied rooms. In order to overcome the challenges described above RoomZoner filters the raw sensor firings and attempt to categorize occupied rooms as being either *transitionally occupied* or *stably occupied*. Ideally, passageway rooms, short-term room usage, and initial room occupancy would be categorized as transitionally occupied while rooms that are being used for longer periods of time, including multi-room usage scenarios would be identified as stably occupied.

The occupancy model used by RoomZoner defines room usage based on rates of sensor firings for transitional occupancy of stable occupancy to start or end. Since the number of sensors differ per room, these rates of firing take the sensor counts into consideration. Thus, for each room of the house, four sets of parameters are defined in order to detect transitional occupancy or stable occupancy starting or ending. Each set of parameters is composed of a firing count, C and a timeout, T . For instance, for the living room to start being transitionally occupied, the number of sensor fired within the timeout defined for starting transitional occupancy divided by the number of sensors in the room has to be greater than the firing count for the starting of transitional occupancy. For transitional occupancy to end the number of sensors fired within the timeout for transitional occupancy ending, normalized by the number of sensors, should be less than the firing count for transitional occupancy ending. Similarly, RoomZoner uses parameters to detect stable occupancy. Transitional occupancies are usually based on shorter timeouts while stable occupancies are based on longer timeouts.

The model is generated by processing historical occupancy data and searching for parameters for each room using an optimization function. The process involves three steps: (i) false positive minimization, (ii) search space generation, and (iii) parameter selection which will be described below.

False Positive Minimization

False positives occur when sensors fire in a room that is not occupied. In order to minimize these occurrences, which could lead to unoccupied rooms being assessed as occupied, RoomZoner aggregates sensor readings from multiple sensors in a room to filter out false firings from any one sensor. As more sensors are aggregated, the possibility of false positives occurring decreases.

Algorithm 1 describes the false positive filtering algorithm RoomZoner utilizes. It takes in raw motion sensor firings, F_R , and outputs aggregated firings, F_A . The filtering algorithm searches for one minute intervals when the number of sensor firings for a particular room is at least half the number of sensors in that room. Each time it finds such an interval, all the sensor firings within that interval are aggregated into a single sensor firings. By enforcing the requirement for at least half the sensors in a room to fire, or a particular sensor to fire at least $n/2$ times for a room with n sensors, the filtering algorithm attempts to minimize instances when a single spurious sensor firing is considered an indication of occupancy. This technique also helps minimize false negatives because for any one minute interval only $n/2$ sensors have to fire for that whole interval to be considered as

Algorithm 1 False Positive Minimization

```

aggregateData = []
windowStartIndex = 1
while windowStartIndex < length(sensorFirings) do
  windowEndTime = sensorFirings(windowStartIndex) + oneMin
  firingsBeforeEnd = sensorFirings < windowEndTime
  firingsInWindow = firingsBeforeEnd(windowStartindex : end)
  if length(firingsInWindow) ≥ numSensors/2 then
    aggregateData.append(firingsInWindow(end))
    windowStartInterval = windowEndInterval + 1
  else
    windowStartInterval = windowStartInterval + 1
  end if
end while

```

having a sensor firing. Thus, residents don't have to constantly keep triggering sensors in order for them to be detected as being present in a room.



Figure 5.3: The solid lines indicate one minute intervals and the dotted lines indicate when sensor data was collected from a room with two motion sensors.

Figure 5.3 illustrates an example of data collected from a room with two motion sensors. In this example, the sensor firings in intervals with a single sensor firing would be eliminated while the firings in intervals with multiple sensor firings would be aggregated into a single firing as shows in Figure 5.4.



Figure 5.4: For a room with two motion sensors, false positive minimization removes the firings from intervals with fewer than two sensor firings and aggregates the firings in intervals with multiple sensor firings.

Search Space Generation

F_A is scanned using moving windows and the number of aggregated firings within the window is compared to thresholds to identify if occupancy began or ended. The following four occupancy parameters are used to define the beginning and ending of occupancy:

1. Occupancy start window size, W_S

2. Occupancy start firing threshold, FT_S
3. Occupancy end window size, W_E
4. Occupancy end firing threshold, FT_E

W_S and FT_S define a frequency, f_o , with which the aggregated sensor firings should occur for the room to be considered as having become occupied and W_E and FT_E define a frequency, f_v , with which the sensors have to fire for the room to be considered as having become vacant. If the observed frequency of sensor firings in F_A is greater than or equal to f_o , meaning there are at least FT_S sensor firings within a window of size W_S , the room is considered to have started becoming occupied at the end of the window with size W_S . If the frequency of sensor firings drops to below f_v , the room will be considered to have become vacant at the end of the window with size W_E . Figure 5.5 shows an example of occupancy start and end being identified with $W_S = 2$ min, $FT_S = 2$, $W_E = 3$ min, $FT_E = 2$.

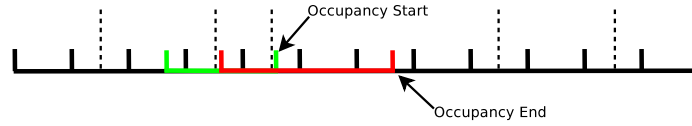


Figure 5.5: The occupancy start window, W_S is shown in green while the occupancy end window, W_E , is shown in red. The times when occupancy would be considered to have started and ended are labeled.

Multiple values for W_S , FT_S , W_E , and FT_E are tried, and the following statistics collected for each set of parameters:

1. False Negative count, FN : number of sensor firings that are ignored
2. Occupancy period, T_O : total amount of time a room is considered occupied
3. Occupancy transitions, OT : number of times a room transitions between being occupied and unoccupied
4. 25th percentile period T_{25} : the minimum T_O for at least 25% of all occupancy durations in the data set

The output of the search space generation process is a set of occupancy parameters, OP , and corresponding statistics. The parameter selection function searches over the statistics to identify parameters that can distinguish stable occupancy from transitional occupancy.

Parameter Selection

In order to identify optimal parameters for occupancy assessment, the parameter selection algorithm first prunes the search space of statistics using the following three thresholds:

1. Maximum number of false negatives, FN_{max}
2. Maximum number of occupancy transitions, OT_{max}
3. Maximum 25th percentile period, $T25_{max}$

Two sets of these three parameters are defined, one for stable occupancy and one for transitional occupancy as shown in Table 5.1. These parameters were selected to address the issues described in Section 5.2.1. In order to minimize short cycling, a longer $T25_{max}$ for stable occupancy was defined and OT_{max} was defined to be small. With these restrictions, FN_{max} was increased to allow the algorithm some flexibility in selecting parameters. For transitional occupancy, false negatives were a concern since the aim of transitional occupancy is to capture occupancy in passageway rooms and during short-term room usage. Thus, the threshold for FN_{max} was set very low and the threshold for OT_{max} was made high since there would be many transitional occupancy events during a day. Finally, a short duration for $T25_{max}$ was defined in order to capture the short periods of time when passageway and short-term rooms are in use.

| | FN_{max} | OT_{max} | $T25_{max}$ (min) |
|--------------|------------|------------|-------------------|
| Stable | 30 | 4 | 30 |
| Transitional | 4 | 30 | 3 |

Table 5.1: Parameters selected for search space pruning reflect the characteristics of stable and transitional occupancy.

Once the search space is pruned, the remaining parameters in OP form the set of *candidateParameters*. This set is searched for the parameters that have the minimum total occupancy time, T_O . This stage of the algorithm outputs two sets of occupancy parameters, one to identify stable occupancy and one to identify transitional occupancy. Algorithm 2 describes the parameter selection process.

Using the parameters selected by Algorithm 2 to process sensor readings for a house over a period of ten days produces the occupancy patterns as depicted in Figure 5.6. It is clear that transitional occupancy captures frequent occupancy changes as detected by aggregated sensor firings, while stable occupancy captures long-term room usage.

Algorithm 2 Parameter Selection

```

for  $i = 1 : \text{length}(\text{params})$  do
  if  $\text{params}(i,1)/\text{numDays} < FN$  and  $\text{params}(i,2)/\text{numDays} < K$  and  $\text{params}(i,3) \geq T_{25}$ 
  then
     $\text{candidateParameters.append}(\text{params}(i,:))$ 
  end if
end for
 $\text{minOccupiedDurationIndex} = \text{find}(\text{min}(\text{candidateParameters}(:,4)))$ 
 $\text{selectedParameters.append}(\text{candidateParameters}(\text{minOccupiedDurationIndex}, 5 : \text{end}))$ 

```

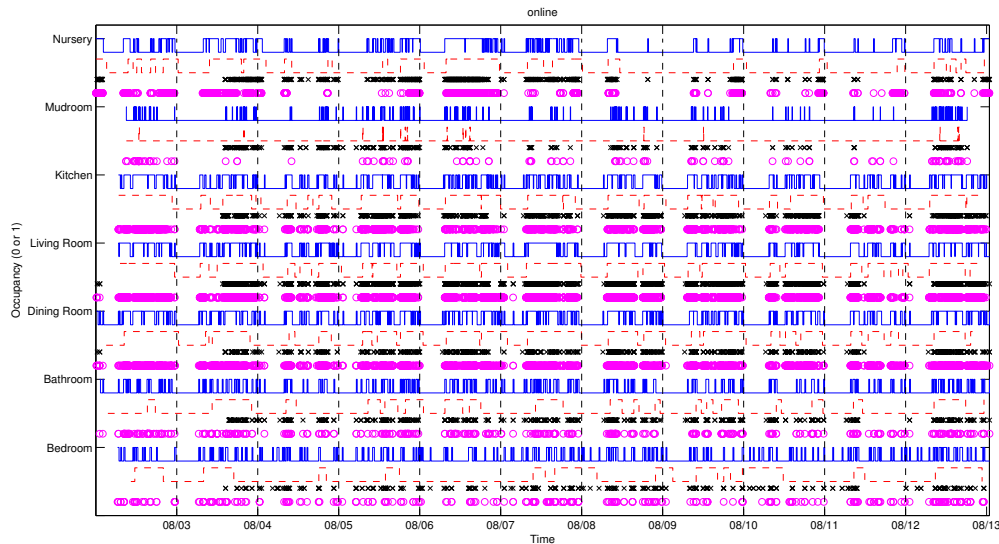


Figure 5.6: The solid line represents transitional occupancy, the dashed line represents stable occupancy, the X's depict the firings of the X10 motion sensors in rooms, and the O's depict the firings of the PIR sensors on doorways.

5.3 Zone Control

After occupancy assessment, the next stage carried out by RoomZoner is hardware control, or *zone control*. This involves deciding on and actuating the HVAC system with appropriate parameters and opening, or closing, the appropriate dampers in order to direct conditioned air through the house. In this section the goals of zone control, the challenges that have to be overcome in implementing it, and the actual implementation are described.

5.3.1 Goals

Since RoomZoner attempts to minimize the energy wasted in conditioning unused rooms, the goal of zoning control is to only condition rooms when they are used. In the ideal case this can be achieved by turning on the equipment conditioning a room when the room is occupied and turning it off when the room becomes unoccupied. Yet, this is complicated when implementing room-level zones using a centralized HVAC system.

The first complication arises due to the thermal dependence between rooms. Houses are not designed with rooms being thermally isolated because it is conditioned by a single piece of equipment and the flow of air from all the rooms towards a thermostat and return vent is desired. Thus, when attempting to zone a house at the room level, it is necessary to take into consideration the thermal interaction between dynamically generated zones.

Using a single piece of equipment, instead of heating and cooling units that can be manipulated for each room individually, such as window air-conditioning units, complicates zoning control because the fine-grained equipment control that is necessary for room-level zoning cannot be easily achieved. A room-level unit cannot be turned on or off as a resident enters or leaves a room and the effect, across multiple rooms, of actuating the HVAC system has to be taken into consideration.

Finally, a zone control algorithm has to make decisions on equipment actuation based on multiple conflicting zone states. For instance, a particular occupied room could be too hot or too cold while another occupied room has to be conditioned, or one room may require stage 2 conditioning, due to it being further away from the setpoint than another room that requires stage 1 conditioning. The zone control algorithm has to decide between these conflicting requirements while attempting to minimize energy consumption without compromising occupant comfort. Due to the above constraints of room-level zoning using a single piece of equipment, a sophisticated zone controller is necessary to achieve the goal of conditioning rooms only when used.

5.3.2 Challenges

As was briefly pointed out above, zone control for room-level zoning using a centralized HVAC system is challenging. This section describes some of the challenges that have to be overcome in implementing the zone controller for RoomZoner.

Interdependence Between Zones

The first challenge is the interdependence between zones which is caused by all the rooms sharing the same network of HVAC ducts. Thus, opening and closing certain ducts, using dampers, has an effect on the airflow through other ducts. For instance, if two rooms are serviced by branching ducts off a common trunk duct, closing off one of the rooms would increase the airflow rate into the other room. Thus, knowing the effect of closing various combinations of dampers is essential in deciding on dynamic zones. A simple solution would be to know the layout of ducts in the house being retrofitted, but this may not always be known by the homeowners. Therefore, a method based on airflow measurements from the registers, which can be easily taken using a handheld airflow meter, is presented.

Minimum Airflow

The centralized HVAC system poses a challenge to room-level zoning and one of the major reasons for this is the minimum airflow requirement for forced-air HVAC systems. These systems rely on forcing air over a condensation coil, transferring heat between the refrigerant and the air, and then delivering the air through a series of ducts to the rooms of a house. HVAC systems are rated for a certain output airflow depending on the operating stage. For instance, the HVAC system in the house where the experiments were carried out produces $830 \text{ ft}^3/\text{min}$ (CFM) of conditioned air when cooling in stage 1, 1200 CFM of air when cooling in stage 2, etc. Ducts are usually properly sized so that most of the air delivered by the fan exits the ducts, which prevents pressure buildup within the ducts, and enables a constant flow of air over the coil. Ducts are also sized to distribute air to rooms depending on the size of the room, with larger rooms having more registers and wider ducts than smaller rooms.

Room-level zoning requires these ducts to be closed which decreases the amount of air that can leave the ducts through the registers. This causes leakage through openings such as insufficiently insulated joints and could even cause pressure buildup within the ducts that slows down the flow of air over the coil. This *back-pressure* could damage the compressor due to insufficient thermal flow between the refrigerant and air causing the refrigerant to not fully vaporize before flowing back into the compressor. Thus, ensuring a minimum airflow out of the registers when deciding on which dampers to close is a challenge that has to be overcome to ensure equipment safety.

Zoning a centralized HVAC system involves closing ducts so that air only flows to a subset of a house. If too many ducts are closed, or ducts are closed in a wrong configuration, back-pressure could build up resulting in energy wastage due to leakage and equipment damage. In order to enable room-level zoning of a centralized HVAC system the buildup of back-pressure has to be taken into consideration when making actuation decisions.

Short Cycling

Manufacturers recommend a minimum time for which an HVAC system should operate at a particular stage before transitioning to a lower stage, for instance transitioning from stage 2 to stage 1 or turning off from stage 1. Transitioning before this minimum threshold increases the wear and tear on the equipment due to it cycling more frequently and doesn't allow the pressure to equalize between cycles. Therefore, in implementing a system that controls the HVAC equipment at a fine granularity, it is essential that the compressor is not short cycled. This adds another factor to be considered when making actuation decisions.

Efficiency Dependence

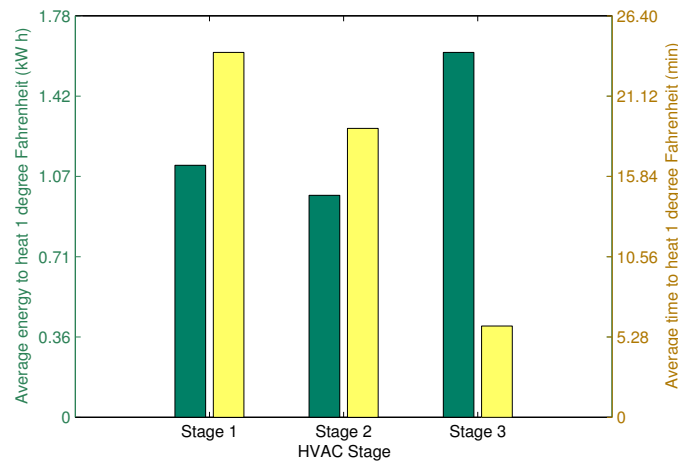


Figure 5.7: Energy efficiency and lag time vary among the multiple stages of HVAC.

The minimum airflow requirement makes selecting an efficient HVAC stage challenging because there is a dependence between efficiency and airflow rate. As Figure 5.7 shows, stage 2 heating is the most energy efficient stage for heating a room by one degree. Yet, as Table 5.2 shows, this stage

produces conditioned air at a higher rate than stage 1 heating. Due to this, more rooms have to be open for stage 2 to be usable. Therefore, a challenge for zone control is making this trade-off between using efficient HVAC stages and minimizing the load on the selected stage so that the rooms being conditioned reach the setpoint faster.

| Stage | Air Output Rate (CFM) |
|--------------|-----------------------|
| Stage 1 Cool | 830 |
| Stage 2 Cool | 1200 |
| Stage 1 Heat | 775 |
| Stage 2 Heat | 1200 |
| Stage 3 Heat | 775 |

Table 5.2: Conditioned Air Output for Different Heating and Cooling Stages.

Thermal Transfer

Thermal transfer between rooms has to be taken into consideration when deciding on rooms to be conditioned at any given time. For instance, in a house with an open floor-plan with the kitchen and living room sharing a large opening between them, attempting to condition the living room and not the kitchen, or vice versa, would cause a wastage in energy due to the large amount of leakage of conditioned air from the conditioned room to the unconditioned room. In such a situation maintaining dependent rooms at a temperature close to the rooms being conditioned would reduce the energy wastage by decreasing the temperature gradient between the rooms. Thus, identifying these inter-dependencies is a challenge that has to be addressed for zone control.

Zone Coordination

The biggest challenge to implementing room-level zoning using a centralized HVAC system is coordinating the conditioning of zones so that energy is not wasted by the compressor constantly being in operation or air leaking between conditioned and unconditioned zones. RoomZoner attempts to minimize the inefficiency by conditioning thermally homogeneous zones together so that the temperature gradient within such zones is relatively small. This would reduce the amount of leakage out of conditioned rooms and minimize the amount of time the HVAC system has to be turned on when an unoccupied room is occupied because it would be close in temperature to the neighboring rooms and, thus, can quickly be brought to the setpoint after which the compressor can be turned off.

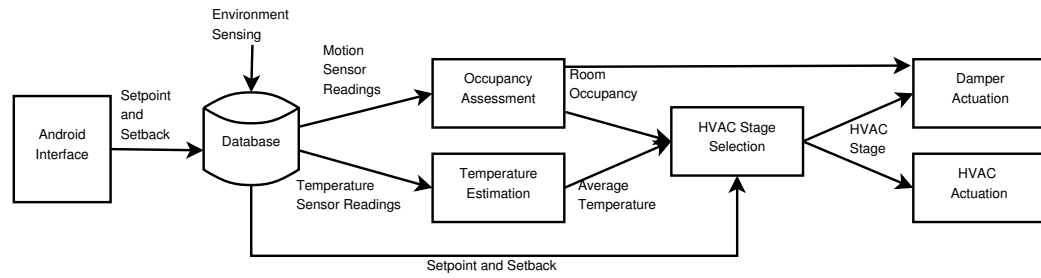


Figure 5.8: The data flow through the zone control algorithm.

5.3.3 Approach

The zone controller used by RoomZoner attempts to maximize the stability of the system by minimizing the number of system changes. Thus, at any decision point RoomZoner attempts to maintain the damper and HVAC equipment at their current state unless changing their state would greatly decrease the estimated energy used, or leaving the system at its current state would considerably affect resident comfort.

Figure 5.8 shows the main modules that encompass the zone controller and the flow of data through the system. All sensor readings, from temperature and occupancy sensors, are written to a database in real-time. The database also holds the setpoint and setback temperature as input by the resident through an Android smartphone-based user interface (Figure 5.9). These values are read from the database by the zone controller and used to assess occupancy, as described in Section 5.2 and estimate an average temperature as will be described in Section 5.3.3. The room occupancy results and average temperature are used to decide upon the HVAC stage to be used when actuating the HVAC equipment. The HVAC stage decided upon, along with the room occupancy assessment results, are used to decide on the dampers to actuate.

Temperature Estimation

There are a number of temperature averages that could be used to control a room-level zoned HVAC system such as the following:

1. Occupied room average, t_O : the average temperature of all rooms in use
2. Conditioned room average, t_C : The average temperature of rooms with open dampers
3. Whole house average, t_H : the average temperature of all rooms in the house

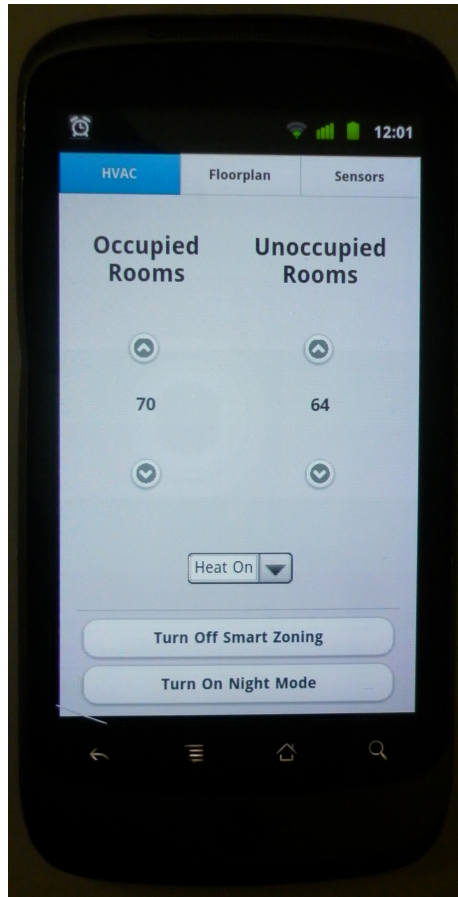


Figure 5.9: Android smartphone-based user interface to HVAC controller.

Yet, controlling a room-level zoned HVAC system using each of these averages individually is not ideal due to the following reasons. The occupied room average, t_O can change every time a resident enters a room or leaves a room. This could cause a high variability in the average temperature used for making control decisions, which could result in *thrashing*: the system state changing frequently. If the newly occupied room hasn't been used recently, it could have drifted far from the setpoint. Adding that room to the set of rooms for which the average is calculated could cause the average temperature to shift drastically resulting in a higher HVAC stage being requested. If the resident leaves the room and enters a room at the setpoint, the average temperature could again change by a large amount causing a lower HVAC stage to be requested or the system to be turned off. Such fluctuations in HVAC control is inefficient in terms of energy usage and could be damaging to the HVAC hardware. Using only conditioned rooms could result in unconditioned rooms drifting far from the setpoint so that when they become occupied a large amount of energy would have to

be expended to condition them to the setpoint. This could be more inefficient than maintaining their temperature closer to the setpoint even when they are unoccupied. Also, bringing these rooms towards the setpoint could take longer, discomforting the residents. Finally, using the average temperature of the whole house would result in the greatest stability in terms of temperature, but this could affect the responsiveness of the system. For instance, the average temperature of the house could be at the setpoint if most rooms are at, or above, the setpoint. But, a particular room could still be below the setpoint. This could be because the room is small and has fewer air vents causing it to receive less conditioned air, it being less well insulated than the other rooms, or it receiving a large amount of sunlight due to its location. A resident entering such a room could be discomforted and the system would be unable to detect it due to the other rooms skewing the average temperature towards the setpoint.

Algorithm 3 Temperature Estimation

```

wholeHouseTempSum = 0
numRooms = 0
for room in rooms do
    wholeHouseTempSum + = room.temperature
    numRooms + = 1
end for
wholeHouseAvg = wholeHouseTempSum / numRooms
conditionedRoomsTempSum = 0
numConditionedRooms = 0
for room in rooms do
    if room.damperClosed == False and room.transitionallyOccupied == False then
        conditionedRoomsTempSum + = room.temperature
        numConditionedRooms + = 1
    end if
end for
conditionedRoomsAvg = conditionedRoomsTempSum / numConditionedRooms
occupiedRoomsTempSum = 0
numOccupiedRooms = 0
for room in rooms do
    if room.stablyOccupied then
        occupiedRoomsTempSum + = room.temperature
        numOccupiedRooms + = 1
    end if
end for
occupiedRoomsAvg = occupiedRoomsTempSum / numOccupiedRooms
if mode == 'cool' then
    averageTemp = min(wholeHouseAvg, conditionedRoomsAvg, occupiedRoomsAvg)
else
    averageTemp = max(wholeHouseAvg, conditionedRoomsAvg, occupiedRoomsAvg)
end if

```

In order to minimize the shortcoming of each of the average temperature calculation methods, RoomZoner uses a hybrid of all three averages to make control decisions. Algorithm 3 shows the hybrid approach adopted by RoomZoner. When heating, RoomZoner uses the maximum of the three temperatures while when cooling it uses the minimum of the three. This approach reduces temperature variance and increases system stability.

HVAC Stage Selection

The HVAC stage selection process take the following inputs and decides upon the stage with which the HVAC should be actuated:

1. Room occupancies
2. Average Temperature
3. Setpoint temperature
4. Setback temperature

RoomZoner uses a hysteresis algorithm similar to the one used by Dual-Zone in order to increase system stability. RoomZoner uses a state machine with three states, excluding the initialization state, instead of four states as used by Dual-Zone. This was because the custom thermostat implemented for RoomZoner allowed individual HVAC stages to be called instead of relying on the manipulation of setpoints on a traditional thermostat as was done by Dual-Zone. Figure 5.10 shows the state machine used by RoomZone to implement hysteresis. The initialization state is defined as -1 , and 0 indicates the HVAC system being turned off. States 1 and 2 correspond to the HVAC stages described in Section 5.3.2. The asymmetric transitions between states, for example the transition from Off to Stage 1 requiring a temperature difference of 0.7 degrees while the transition from Stage 1 to off requiring a difference of only 0.5 degrees, provides a 0.2 degree dead-band to prevent rapid fluctuations between the two states.

Algorithm 4 shows how hysteresis is used to select an HVAC stage. RoomZoner compares the average temperature generated by Algorithm 3 to the setpoint if the house is occupied. Otherwise, it compares the average temperate to the setback.

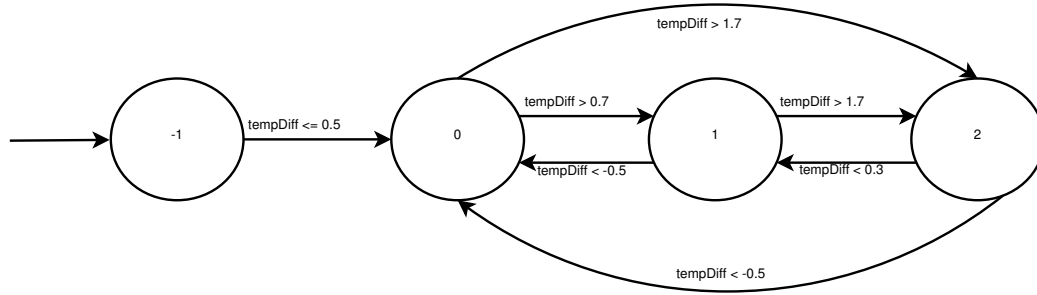


Figure 5.10: The finite-state machine which provides hysteresis during the stage selection process.

Algorithm 4 HVAC Stage Selection

```

if numOccupiedRooms > 0 then
  tempDiff = averageTemp − setpoint
else
  tempDiff = averageTemp − setback
end if
if mode == 'heat' then
  tempDiff = −tempDiff
end if
if hvacStage == −1 then
  if tempDiff ≥ 1.7 then
    hvacStage = 2
  else if tempDiff > 0.5 then
    hvacStage = 1
  else
    hvacStage = 0
  end if
else if hvacStage == 0 then
  if tempDiff > 1.7 then
    hvacStage = 2
  else if tempDiff > 0.7 then
    hvacStage = 1
  end if
else if hvacStage == 1 then
  if tempDiff > 1.7 then
    hvacStage = 2
  else if tempDiff < −0.5 then
    hvacStage = 0
  end if
else
  if tempDiff < −0.5 then
    hvacStage = 0
  else if tempDiff < 0.3 then
    hvacStage = 1
  end if
end if

```

Damper Actuation

Damper actuation involves selecting the dampers to be opened and closed and sending the appropriate commands to the damper control circuitry. Since the zone controller does not use room-level temperature control all rooms that are occupied have their dampers open. The HVAC stage selected by Algorithm 4 determines the minimum airflow into the rooms in order to ensure equipment safety. The damper actuation module attempts to ensure this minimum airflow by strategically opening additional rooms as needed so that there is minimum impact to the efficiency with which the occupied rooms are conditioned. The decision that has to be made is on what additional rooms, called *dump rooms*, that comprise the *dump zone* have to be opened, if they are closed. This decision is based on ensuring the safety of the HVAC equipment by minimizing the chance of back-pressure buildup due to too many dampers being closed.

Algorithm 5 is used during zone control to select the rooms that comprise the dump zone. Whenever there are insufficient rooms occupied in order to ensure sufficient airflow out of the ducts, the dump zone selection algorithm is called. This algorithm selects dump zones based on the estimated benefit, in terms of thermal transfer, to the occupied rooms rather than the actual temperatures of the potential dump rooms. Also, the algorithm prefers rooms that have been occupied in the near past as dump rooms. These approaches to dump zone selection ensures the zone controller is able to achieve its goal of maximizing system stability by minimizing state changes.

The first phase of dump zone selection is separating the rooms into active rooms, those room that require conditioning due to either being stably or transitionally occupied or being beyond the setback temperature, and dump candidates. Next, all combinations of dump candidates are evaluated in order to identify the set of rooms that have the greatest positive impact on the goals of zone control.

The first step in dump candidate evaluation is estimating the total airflow out of the ducts with a particular set of dump rooms. The airflow calculator provides a conservative estimate of the expected airflow when a particular set of registers is closed. This estimation is based on empirical values of airflow collected with a airflow meter. Two values of airflow are collected for each register: *openAirflow* and *closedAirflow*. *openAirflow* is the volume of air output from a register when all registers are open, while *closedAirflow* is the volume of air output from a register when it is closed while all other registers are open. these values are used as approximations of the airflow out of a register when it is opened or closed. They are lower bounds since if any other register is closed

concurrently, these values are expected to increase. The total airflow is calculated by adding together either a closedAirflow or an openAirflow value for each register in the house depending on whether the register is closed or open. This sum provides a lower bound for the amount of air expected to leave the ducts. This value being beyond the safety limit of the HVAC system would ensure the safety of the hardware. A lower bound calculation was used instead of an accurate measurement of the airflow due to the large number of measurements necessary to build such a model. For the residence where the system was implemented, which has 13 registers, $2^{13} = 8192$ measurements would have to be taken to build a complete model. Using the lower bound allows the model to be built within a few hours.

The estimated airflow is used to eliminate dump candidates that would not help with ensuring a safe volume of airflow out of the system. In addition to ensuring that this safety limit is met, RoomZoner also verifies that no room in the set can be removed and still have a safe airflow out of the system using the minimum airflow change which is the minimum difference in airflow from a room in the set being opened and closed. This minimum airflow change is compared to the difference between the airflow estimate and the safe airflow to ensure that the change is greater than the difference and therefore the room has to be opened.

The final step of damper actuation is optimizing over the search space of dump candidates that passed the airflow check. The optimization is done over two parameters: *zone energy* and *damper changes*. Zone energy is an estimate of the thermal impact on occupied rooms by a particular dump zone and damper changes is the number of rooms that have to be changed from currently being opened to closed, or closed to opened. Zone energy is calculated using voltage values that are obtained from a circuit model of airflow in a house that we describe in the next subsection. This model assume rooms are wires with resistors between them and the air coming out of the dampers are current sources. Give a set of current sources, as estimated airflows from the registers, the model provides voltages for each room. These voltages describe temperature changes in rooms with higher voltages indicating a greater temperature change. Thus, the optimization function attempts to search for dump zones that have the greatest impact on the voltage of the occupied rooms, indicating the possibility of the greatest positive impact on those rooms in terms of thermal transfer. With this approach we address the thermal transfer challenge described in Section 5.3.2. In addition to the zone energy, we also attempt to minimize the number of damper changes that have to be made to achieve the dump zone. This ensures that rooms that were recently occupied are preferred

as dump candidates since their dampers are more likely to be open and also minimizes the number of state changes necessary, as the zoning controller is trying to achieve. Preferring recently occupied rooms helps ensure that rooms that are used in multi-room occupancy scenarios are kept close to the setpoint. This also helps ensure that the temperature of rooms that are more likely to be occupied in the future, due to temporal locality, are not allowed to drift far from the setpoint.

Thermal Model

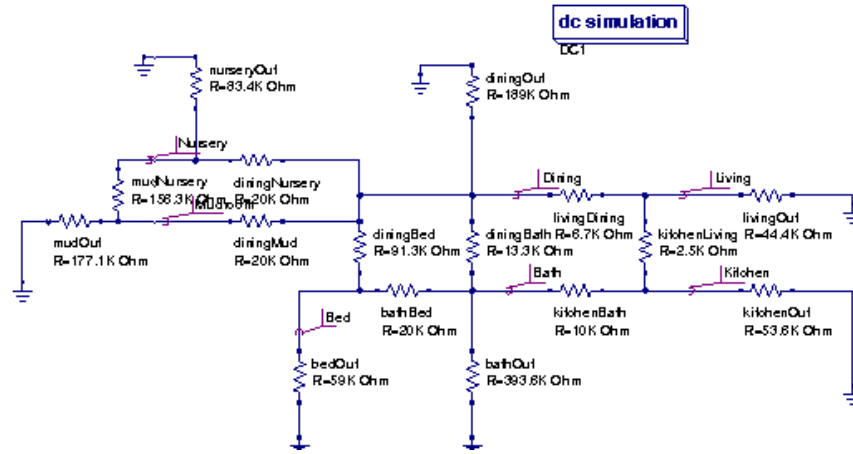


Figure 5.11: Circuit model of house generated in Qucs circuit simulator.

A circuit model (Figure 5.3.3) of the house is generated using the Quite Universal Circuit Simulator (Qucs) [138]. The model is based on the assumption that the total resistance between the house and the external environment is $1\text{ M}\Omega$. This total resistance is distributed across all rooms based on the total surface area of external walls and ceilings in a room (Table 5.3).

The $1\text{ M}\Omega$ resistance is distributed across the rooms in an inverse proportion to the external surface area using the following set of equations:

$$\begin{aligned}
 R_L + R_K + R_{Ba} + R_{Be} + R_M + R_N + R_D &= 1\text{M}\Omega \\
 554.03R_L &= 426.75R_K = 84.6R_{Ba} = 368R_{Be} = \\
 144R_M &= 233.75R_N = 275.25R_D
 \end{aligned}$$

Solving the set of equations above give the external resistances in Table 5.3.

Further, we assume a 25°F temperature difference between the interior and exterior of the house

based on observations of an ability to maintain the temperature in the house at 70°F when the external temperature is 45°F. This 25°F is translated to a 25V potential difference between the interior and exterior, V_{ext} and current flows out of each room, which translate into heat loss to the external environment, are calculated using $I = V_{ext}/R$ where I is the current through external walls and ceilings and R is the resistance between each room and its exterior from Table 5.3. The heat loss, in terms of current is given in Table 5.3.

| Room | Area (ft ²) | Resistance (K Ω) | Current(μ A) |
|-------------|-------------------------|--------------------------|-------------------|
| Living Room | 554.03 | 69.9 | 357.7 |
| Kitchen | 426.75 | 53.8 | 464.7 |
| Bathroom | 84.6 | 352.4 | 70.9 |
| Bedroom | 368 | 81 | 308.6 |
| Mudroom | 144 | 207 | 120.8 |
| Nursery | 233.75 | 127.5 | 196.1 |
| Dining Room | 275.25 | 108.3 | 230.8 |

Table 5.3: Area of, resistance of, and current through external surfaces.

We then calculate all the currents through the circuit using Kirchhoff's equations given a current input into the Kitchen, I_{KI} , which represents airflow into the kitchen. The following set of equations are solved to get all currents in the circuit in this steady-state scenario. The subscripts on the current variables, I , indicate the source and destination of the current so that I_{KL} is a current flowing from the kitchen to the living room. The only variable that violates this convention is I_{KI} which is the input current source into the kitchen.

$$I_{KI} = 464.7\mu + I_{KL} + I_{KBa} \quad (5.1)$$

$$I_{KL} = 357.7\mu + I_{LD} \quad (5.2)$$

$$I_{KBa} = 70.9\mu + I_{BaBe} - I_{DBa} \quad (5.3)$$

$$I_{LD} = 230.8\mu + I_{DBa} + I_{DBe} + I_{DM} + I_{DN} \quad (5.4)$$

$$I_{DBa} = 70.9\mu + I_{BaBe} - I_{KBa} \quad (5.5)$$

$$I_{DBe} = 308.6\mu - I_{BaBe} \quad (5.6)$$

$$I_{DM} = 120.8\mu + I_{MN} \quad (5.7)$$

$$I_{DN} = 196.1\mu - I_{MN} \quad (5.8)$$

$$I_{BaBe} = 308.6\mu - I_{DBe} \quad (5.9)$$

$$I_{MN} = 196.1\mu - I_{DN} \quad (5.10)$$

Algorithm 5 Dump Zone Selection

```

for roominrooms do
  activeRooms = []
  dumpCandidate = []
  if room.stageRequest > 0 or room.transitionallyOccupied then
    activeRooms.append(room)
  else
    dumpCandidates.append(room)
  end if
end for
dumpSets = []
for dumpSetincombinations(dumpCandidates) do
  airflow = calculateAirflow(activeRooms, dumpSet)
  minFlowChange = INF
  for roomindumpSet do
    if room.openAirflow < minflow then
      minFlowChange = room.openAirflow - room.closedAirflow
    end if
  end for
  if airflow > safeAirflow and minFlowChange > airflow - safeAirflow then
    dumpSets.append(dumpSet)
  end if
end for
bestEnergy = -INF; bestChanges = 0; bestDumpZone = []
for dumpSetindumpSets do
  inactiveRooms = rooms
  for roominactiveRooms do
    inactiveRooms.remove(room)
    activeZoneEnergy += voltages(hvacStage, dumpSet, room)
  end for
  damperChanges = 0
  for roomindumpSet do
    inactiveRooms.remove(room)
    if room.damperClosed then
      damperChanges += 1
    end if
  end for
  for roomininactiveRooms do
    if room.damperClosed == False then
      damperChanges += 1
    end if
  end for
  if activeZoneEnergy - damperCost * damperChanges > bestEnergy - damperCost *
  bestChanges then
    bestDumpZone = dumpSet
    bestEnergy = activeZoneEnergy
    bestDamperChanges = damperChanges
  end if
end for
dumpZone = []
for roominbestDumpZone do
  dumpZone.append(room)
  estimatedAirflow = calculateAirflow(activeRooms, dumpZone)
  if estimatedAirflow > safeAirflow then
    returndumpZone
  end if
end for

```

5.4 Preliminary Implementations and Lessons Learned

The initial implementation of occupancy assessment involved a simple threshold based approach. For a room with N sensors, if $N/2$ sensor firings were recorded within a one minute interval, the room was considered to be occupied. This was based on the assumption that if at least half the sensors in a room fired within a short period, an actual occupancy should have been detected within the room.

While this approach proved sufficiently reliable to detect room occupancies, it could not differentiate between rooms being actually occupied or sensors being triggered as people passed through rooms. This causes the occupancy assessment algorithm to classify occupancy as either stable or transitional based on sensor firing characteristics.

A number of HVAC stage selection and dump zone selection algorithms were implemented before settling on the algorithms described above. In the following subsections some of these approaches are described.

HVAC Stage Selection

RoomZoner uses the average temperature to select an HVAC stage. Before settling on this implementation, two alternative HVAC stage selection strategies, as described below, were considered.

Majority room request. In this scheme, each room is given a vote to decide on an optimal HVAC stage depending on its temperature and occupancy. Hysteresis is used on a per-room basis using the algorithm depicted in Figure 4.8. Once all the rooms cast their votes, the majority from among the votes is used to control the HVAC system. For instance, if five of the seven rooms requested stage 1 cooling while the other two rooms requested stage 2 cooling, the HVAC would be actuated in stage 1.

In most instances very few rooms are far enough from the setpoint to request stage 2 and, therefore, the HVAC system operates in stage 1 almost exclusively when it is turned on. This results in rooms that require stage 2 heating or cooling to not receive the appropriate volume of conditioned air and, thus, take a long time to reach the setpoint or not be able to achieve that temperature.

Max room request. Due to the shortcomings of the majority room request HVAC stage selection approach, the stage selector was modified to use the maximum stage request. With this

approach, if any room requires stage 2 this stage would be used to condition all the rooms. Such an approach is based on the assumption that rooms requiring a higher stage could reach the setpoint within a reasonable time while rooms requiring a lower stage would reach the setpoint faster than with the lower stage, thus saving energy.

What was observed when executing the system with the max stage request HVAC stage selection policy was many rooms overshooting the setpoint temperature due to them requiring less conditioned air than that provided with stage 2 heating or cooling. This results in discomfort for residents due to rooms being too cold or too hot.

Dump Zone Selection

RoomZoner selects dump zones based on their estimated influence on the active rooms using a simple model of thermal flow through the house. Before implementing this dump zone selection technique, three dump zone selection algorithms were considered. The first one was based on room temperatures, the second was based on a pre-defined selection priority, and the third was based on room connectivity.

Temperature-based dump zone selection. In this approach, for each room that was not occupied RoomZoner calculates the difference between the temperature in the room and the setpoint. It then sorts these dump candidates based on temperature difference so that rooms further away from the setpoint get a higher priority of being selected as dump rooms than rooms closer to the setpoint. This approach was based on the intuition that maintaining the average temperature of the unoccupied rooms at close to the setpoint as possible would minimize the time required to heat, or cool, these rooms when they were occupied.

Execution of RoomZoner using this dump zone selection approach caused dump zones to constantly fluctuate. For instance, in a particular iteration of RoomZoner if the bathroom and a bedroom were dump candidates with only one of the two required for the dump zone and the bathroom was further from the setpoint than the bedroom, the bathroom would be conditioned and the bedroom left unconditioned. In the next iteration, due to being conditioned the bathroom would be closer to the setpoint than the bedroom causing the bedroom to be selected as the dump room and the bathroom to be left unconditioned. Such an approach has two drawbacks. The first is the constant alternating between rooms could result in a room never being sufficiently heated or cooled to approach the setpoint. Each time a room alternates from being a dump room to being left

unconditioned, the energy expended in heating or cooling that room could be lost due to leakage. The second shortcoming of this approach is the constant opening and closing of active registers or dampers as rooms are added and removed from the dump zone. This could reduce the lifetime of these dampers, both in terms of energy if they are operating on batteries, as well as the number of mechanical actuations they are capable of sustaining.

Priority-based dump zone selection. In the next attempt at dump room selection, a pre-defined priority scheme with which rooms are added to the dump zone was tried. The following priority scheme was utilized:

- Transitionally occupied rooms have a higher priority than occupied rooms.
- Rooms with open dampers have a higher priority than rooms with closed dampers.
- Rooms further from the setpoint have a higher priority than rooms closer to the setpoint.

This approach to dump zone selection was aimed at minimizing thrashing without compromising occupant comfort. By preferring transitionally occupied rooms to unoccupied rooms, rooms that are occupied, even temporarily, are kept at a comfortable temperature. Transitionally occupied rooms are also more likely to be stably occupied than rooms that are unoccupied for long periods of time. Thus by maintaining them at close to the setpoint, they can be quickly and efficiently conditioned to the setpoint in the event that residents begin using them. By preferring rooms with dampers already opened to rooms with closed dampers, the number of transitions between opening and closing dampers is minimized, increasing system stability. Finally, selecting rooms further from the setpoint than those closer to the setpoint ensures no room will drift very far from the setpoint before it is conditioned.

Priority-based dump zone selection resulted in very stable dump zones with minimal thrashing, yet it also caused certain rooms to not be conditioned for long periods of time due to being low on the priority list. When such rooms were occupied, a large amount of energy and time was required to condition it back to the setpoint, and occupants were uncomfortable for long durations of time while the room warmed up or cooled down.

Connectivity-based dump zone selection. In the third attempt at a dump zone selection algorithm, it was decided to prioritize neighboring rooms when selecting dump zones. This approach was based on the intuition that neighboring rooms would have the greatest influence on occupied rooms. For instance, if a neighboring room is not conditioned, the temperature difference between it

and its occupied neighbor would be large resulting in a greater thermal transfer between the occupied room and the neighboring room. By conditioning neighboring rooms, this thermal gradient could be minimized reducing the leakage. Also, leakage from a neighboring room into an occupied room could benefit the occupied room.

Connectivity-based dump zone selection minimized fluctuations between dump rooms since occupied rooms do not change as frequently. Due to the fact that rooms neighboring occupied rooms are more likely to be occupied in the near future, and these rooms are selected as dump rooms, the problem of a person entering an unoccupied room and it being far from the setpoint was mitigated. Yet, it was observed that this approach resulted in less conditioned air being provided to occupied rooms. Airflow measurement experiments indicated that closing dampers and registers within a branch of the duct system increased the pressure within that branch and thus forced more air out of the open dampers in that branch. Since neighboring rooms are more likely to share a common branch of the HVAC ducts, opening neighboring rooms resulted in a lower volume of conditioned air entering occupied rooms. The model of thermal flow attempted to capture this phenomenon and suggest rooms that would have the greatest positive impact on occupied rooms.

5.5 Software Implementation

In a similar approach to case study 1, RoomZoner was implemented using Python. Some of the algorithms used in the Python implementation are presented above. Similar to case study 1, the RoomZoner software was, also, written in MacroLab in order to evaluate the ease with which an application written in MacroLab could be evolved as its functionality is extended. The code in Figure 5.12 shows the high level program logic. *assessOccupancy*, *calculateAverageTemperature*, and *selectDumpZone* are functions that can easily be written in MacroLab.

```

1 RTS = RunTimeSystem();
2 weatherdirect = RTS.getMotes('type', 'weatherdirect');
3 tempSensors = SensorVector(weatherdirect, 'temperature');
4 x10 = RTS.getMotes('type', 'X10');
5 motionSensors = SensorVector(x10, 'motion');
6 motionSensorIDs = uint8([8 1 9], [2 6], [10 5], [4], [7 11], [12 14], [3 15]);
7 tempSensorIDs = uint8([1 3], [2], [6 7], [4 9 11], [12 13], [5 14], [8 10]);
8 damperIDs = uint8([3 5], [1 2], [8 4], [10], [11 15], [12 14], [5 7]);
9 nightStart = [0 0 0 2 0 0];
10 nightEnd = [0 0 0 7 0 0];
11 curState = 'On'
12 every(60000)
13     mode = dbRead('zoning', 'mode', 'latest')
14     damperState = dbRead('zoning', 'damper', damperIDs)
15     motionVals = motionSensors.sense();
16     tempVals = tempSensors.sense();
17     [stablyOccupied, transitionallyOccupied] = assessOccupancy(motionVals,
        motionSensorIDs);
18     aTemp = calculateAverageTemperature(tempVals, stablyOccupied,
        transitionallyOccupied, damperState)
19     if length(stablyOccupied) > 0 || length(transitionallyOccupied) > 0
20         SP = dbRead('zoning', 'setpoint', 'latest');
21     else
22         SP = dbRead('zoning', 'setback', 'latest');
23     if mode == 'heat'
24         deltaTemp = SP - aTemp;
25     else
26         deltaTemp = aTemp - SP;
27     curState = getNextState(curState, deltaTemp);
28     if curState > 0:
29         dumpRooms = selectDumpZone(curState, stablyOccupied, transitionallyOccupied
        );
30         hvacActuate(mode, curState, [stablyOccupied dumpRooms]);
31     else
32         hvacActuate(mode, curState, stablyOccupied);
33 end
34 end

```

Figure 5.12: MacroLab implementation of RoomZoner.

5.6 Evaluation

RoomZoner is evaluated in the same house used to evaluate day/night zoning by alternating between room-level zoning and whole house conditioning. In the following subsections the energy measurements of RoomZoner is compared to whole house conditioning and the lessons learned by implementing room-level zoning in MacroLab are discussed.

5.6.1 Zoning Evaluation

To evaluate RoomZoner the control of the HVAC system is alternated between single-zoned whole house control and room-level zoned control over a 42 day period, such that each system ran every other day. This experimental procedure was selected to minimize the effect of changing weather patterns on energy statistics. The experiments with alternating RoomZoner and whole house conditioning were carried out over a period of four months during the winter. For this evaluation we excluded the data from days when the HVAC did not turn on due to the temperature never being below the setpoint and days when there were clear data loss due to unusually low energy consumption values. From the remaining days we extracted 21 days worth of data for each system so that for each month RoomZoner and whole-house control had the same amount of data. This was done to ensure fairness by minimizing the effect of weather when the data for one system is mostly from the coldest part of the winter while the other system's data is from milder days. The energy consumed by the HVAC system was monitored using The E-Monitor total home energy management system [139]. Figure 5.14 shows the energy consumed in conditioning a house using RoomZoner and whole house conditioning. This graph indicates that whole-house conditioning consumed 14.4% more energy than the prototype implementation of room-level zoning, on average.

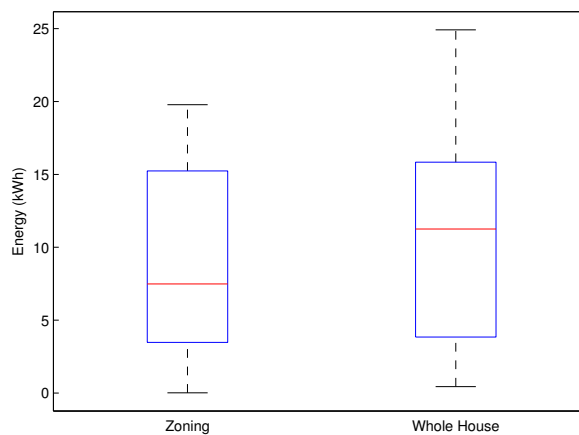


Figure 5.13: The implementation of room-level zoning uses 14.4% less energy than whole house heating on average.

The energy savings for RoomZoner cannot be compared to the savings for day/night zoning due to three reasons. The first is the day/night zoning experiments being conducted during the summer

when the HVAC system was used for cooling while the RoomZoner experiments were conducted during the winter. The second is that the day/night zoned system was compared against a standard residential thermostat controlling the whole house conditioning while RoomZoner was compared against a custom whole house conditioning system that used the same infrastructure as RoomZoner. In whole house mode the thermostat opens all dampers and attempts to maintain the average temperature of all the temperature sensors at the setpoint, in essence treating the whole house as a single zone. This is a fairer comparison of the two systems since the only variable changing between the two is the number of zones which changes from the dynamic number of zones generated by RoomZoner to a single zone. When comparing against a residential thermostat the algorithm used to maintain the temperature changes as well as the number of sensors used to monitor temperature changes from the number deployed across the house to the single sensor at the thermostat. The third is the upgrade of the energy monitoring system from the TED, which monitored power for all appliances from a single point, to E-Monitor which monitored each circuit individual, thus, being able to isolate the power consumption of the HVAC system more accurately.

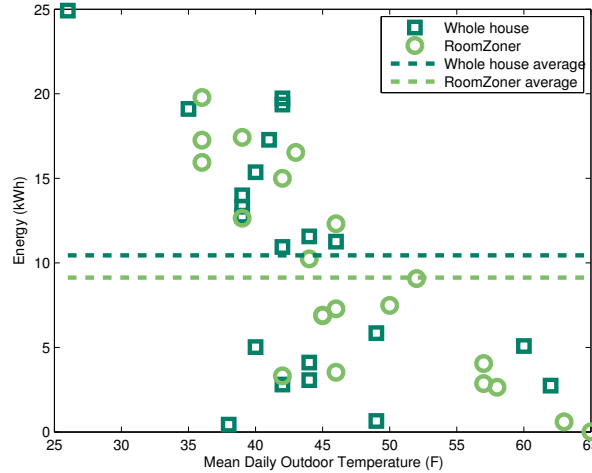


Figure 5.14: The dotted lines indicate the average energy used over the experimental period.

Figure 5.14 shows the actual energy consumption for each day as a scatter plot, with the average temperature of for that day on the x-axis, the energy consumed on the y-axis, and the control algorithm shown as the color of the scatter point.

5.6.2 Macroprogramming Discussion

Extending the Python implementation of day/night zoning to implement RoomZoner required the introduction of a number of functions to assess occupancy and dynamically change zones based on room occupancy and temperature changes. This increased the code size from about 500 lines of code to over 1000 lines of code. Extending the MacroLab implementation of day/night zoning to RoomZoner saw a similar doubling in code size from about 200 lines of code to about 500 lines of code.

Unlike case study 1, the RoomZoner macroprogram cannot be optimized using in-network data processing or data fusion. This is because the zones dynamically vary and therefore temperature and motion information from all the rooms are required to make control decisions. In a large, multi-hop network in-network data aggregation could be used to aggregate the sensor readings within each room in order to calculate an average room temperature which is passed up to the base station, but this optimization doesn't provide as much savings in terms of messages passed as afforded in the day/night zoning scenario. Thus, due to the need for more data as the application complexity increases, the flexibility afforded to MacroLab to perform implementation optimizations decreases. Table 5.4 summarizes the applicability of MacroLab optimizations for Dual-Zone and RoomZoner.

| Optimization | Dual-Zone | RoomZoner |
|----------------------------|-----------|-----------|
| In-network data processing | Yes | No |
| In-network aggregation | Yes | Yes |
| Data fusion | Yes | No |

Table 5.4: As the application complexity increases the freedom for MacroLab optimizations decreases.

While the benefit of macroprogramming abstractions in terms of implementation optimization decreases as the application complexity increases, they are still valuable in implementing CPS applications. Their main value they provide is by decreasing the burden on the application developer by maximizing code re-usability and providing a uniform abstraction for data and hardware accesses. In the case of MacroLab all data within the network is presented as vectors and all hardware actuations are enabled by function calls. If the sensors or actuators being used have been previously used for an application developed using MacroLab, hardware drivers that provide the actuation function and present the sensor data as vector will be available. If the application is using a new set of hardware, the drivers would have to be written the first time after which all future applications could be written assuming the availability of vectors with sensor data and function to perform actuation.

5.7 Conclusions

Case study 2 demonstrated the limitation of macroprogramming abstractions, in terms of code optimization, as application complexity increases. While the amount of optimizations is limited, the simple high-level programming abstraction provided by macroprogramming languages makes developing complex CPS applications easier. Macroprogramming abstractions allow application developers to focus on describing the application logic as they envision it without being burdened with collecting the data from the sensors or being concerned with the passing of messages within the network. Thus, useful CPS applications such as RoomZoner can easily be developed.

RoomZoner enabled simple COTS sensors and actuators to be used to retrofit a centralized HVAC system and dynamically change zones based on occupancy. Such dynamically altering zones resulted in a 14% energy savings as compared to the whole house being conditioned as a single zone. RoomZoner is the first system to enable a centralized HVAC system to be zoned in response to room occupancy. Incorporating prediction, as described in the next chapter, could result in even greater energy savings for such a system.

Chapter 6

SmartZone: Predictive Zoning

Case study 3 extend the zoning system from reacting to changes in occupancy and temperature, as described in Chapter 5, to being able to predict occupancy and temperature. In this case study models that could be used to predict the occupancy of rooms and the effect of HVAC actuations on room temperatures are presented. This chapter also presents an approach to using these models for predictive control of an HVAC system, but to-date have not implemented or evaluated such a system due to it requiring another one year study. Instead, we implement the core components of a predictive zoning system: a predictive thermal model and a predictive occupancy model, and describe how they could be used to predictively zone an HVAC system.

6.1 Predictive Thermal Model

In order to implement a predictive zoning controller the effects of opening or closing air vent registers on room temperatures have to be predicted. Making such a prediction is complicated by the fact that weather has a larger effect on room temperatures than the settings of air vent registers, making it hard to isolate the influence of the HVAC system. A technique for dynamically estimating the heat load due to weather on room temperatures and subtracting it out in order to predict the effect of the HVAC system more directly is presented.

6.1.1 Introduction

A predictive zoning controller needs to be able to predict the effect of opening or closing registers on the room temperatures. Thus, this section describes and evaluates techniques to learn and predict the effect of opening or closing each vent register, in a set of R air vent registers, on the temperature at each sensor, in a set of T temperature sensors placed within a house.

The main challenge to modeling the thermal characteristics of a house is the effect of weather on the indoor temperature. For instance, wind, solar gain, and outdoor temperature have a greater influence on indoor temperature than any individual air vent register. It is difficult to build a model that completely captures the effect of weather on indoor temperatures because outdoor weather conditions constantly change and rarely repeat. The difficulty of attributing the influence on weather conditions on indoor temperature makes it difficult to isolate the effect of the state of any particular air vent register on the indoor temperature.

This problem is overcome by modeling the indoor temperature in two stages. In the first stage, the rate of heat gain or loss due exclusively to outdoor weather conditions is measured. This stage is modeled with data collected when the HVAC system is off using a linear function of current temperature. Then, when the HVAC system is turned on, the *change* in the rate of heat gained or lost in a room due to the conditioned air provided by the HVAC system is measured. This change is expected to be constant throughout the year because the HVAC system always outputs the same amount of conditioned air. Thus, the HVAC effects are isolated by learning and subtracting out a dynamic estimate of weather effects over long periods of time.

This section describes three iterations of a thermal model and analyze its accuracy in terms of predicting the effect of opening and closing various combinations of registers with a centralized HVAC system. Ten-fold cross validation over three weeks of data sampled over three months is used to demonstrate that even with the simplest model temperatures can be predicted to within two degrees 30 minutes into the future. A 30 minute time window is considered because longer time windows are not beneficial when making HVAC control decision. Even the simplest of the three models presented in this section is able to provide this level of accuracy allowing temperature prediction to be incorporated into an HVAC zoning controller easily and without much computation overhead.

6.1.2 Background and Related Work

In order to evaluate the energy efficiency of buildings and implement building energy control strategies, a number of methods have been proposed to model the thermal performance of building envelopes. Well mixed models [140, 141], thermal bridge models [142], mixed convection heat transfer models [143], and CFD models [144, 145] are just a few examples of temperature response models that were developed for buildings. These techniques rely on modeling the building in detail using software such as DOE-2 [146] and COMIS [147] which are then used to predict the thermal flow across zones. This approach is infeasible for a system deployed by a non-expert homeowner since modeling a house in detail is not easy even for experts.

Mozer et al. [130] use an RC model of the Neural Network House, where their Neurothermostat is evaluated. This model assumes that the inside, and outside, of the house are at uniform temperatures, the walls have a thermal resistance, R , and thermal capacitance, C , the entire wall mass is at the inside temperature, and the heat input to the inside is Q when the furnace is running or zero otherwise. Using these assumptions, and constants for R and C determined from the architectural properties of the house, the thermal model was used to predict the future indoor temperatures as a function of the outdoor temperature and the furnace operation.

6.1.3 Problem Definition

The temperature prediction problem is defined by a set of air vent dampers D and a set of temperature sensors T that are dispersed across a house (Figure 4.9). The dampers can be opened or closed, determining if conditioned air is delivered directly into a room. Due to the lack of thermal isolation between rooms, even if the air vent dampers of a room are closed, its temperature could still be affected by the HVAC system due to leakage from neighboring rooms. The temperature sensors monitor the temperatures at different points throughout the house. Figure 6.1 shows the readings at the twelve temperature sensors in the deployment during a day with all air vent dampers open. As the figure shows, the HVAC system being off (blue) causes drops in temperature while the HVAC system being on (red) usually causes temperature increases. The goal is to learn and predict these effects on the temperature sensors when different sets of air vent dampers are opened and closed. In other words, the thermal model is attempting to answer the question *“What effect does each register being open have on the reading of each temperature sensor?”* Being able to make such a prediction

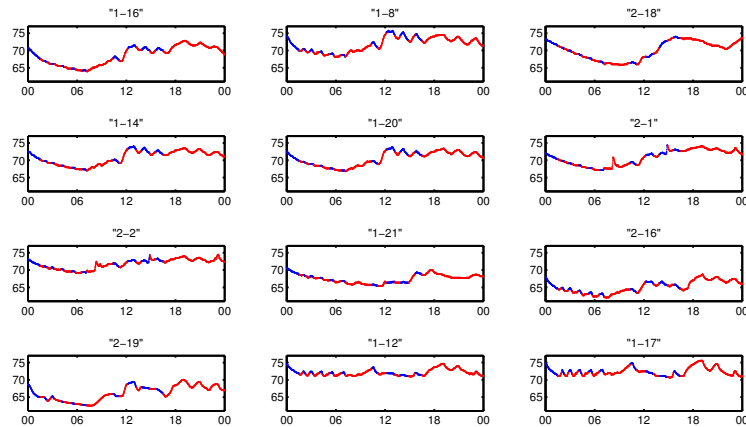


Figure 6.1: The effect on temperature sensors, within a 24-hour period, of the HVAC system being on (red) and off (blue) when heating with all air vent dampers open. The locations of the twelve sensors are presented in Figure 4.9

allows the implementation of a finer-grained automated zoning controller that can dynamically alter zones within a single floor to maintain occupied rooms at a comfortable temperature while allowing unoccupied rooms to drift. Yet, answering this question is difficult due to the effect of the weather on the internal temperature of houses. Wind, solar gain, outdoor temperature, and other weather conditions have a much greater influence on indoor temperature than the conditioned air provided by an HVAC system. These weather conditions constantly change, and rarely repeat, therefore including it as part of a model is impossible without greatly increasing the complexity of the model. But, ignoring the effect of weather on internal temperature makes it impossible to isolate the effect of a particular register on a temperature sensor. Thus, a secondary question that this section attempts to answer is *“Can the effect of dampers on temperature sensors be learned without knowing the weather during the training phase?”* In other words, the model attempts to capture the effect of the weather on the temperature sensor readings while ignoring the actual weather conditions, such as the external temperature or the position of the sun.

There have been a number of approaches proposed for learning the thermal response of buildings in order to control HVAC systems efficiently [148–153]. Yet, these approaches require a large amount of data or sophisticated sensors that will hinder the goal of developing a cheap and easy to install retrofit to enable room-level zoning of existing centralized HVAC systems.

6.1.4 Experimental Setup

The room-level zoning system described has been deployed in a single-story, 8-room, 1,200-square-foot residential building. A model of the home is shown in Figure 4.9. The hallway and porch are depicted, but not included within the analysis because of the inability to actuate temperature within these regions. The HVAC system setup is overlaid in order to show the position of vents, ducts, and the central air handler.

Figure 4.9 shows the deployment from which data for evaluating the thermal model was collected. Twelve temperature sensor deployed across the house and air vent registers that are remotely actuable are used to collect data over a three month period. Three weeks of the collected data was used for the analysis presented in this case study.

6.1.5 Model of Temperature Dynamics

The parameters of the model include the position of the damper, temperature, system status, and time. These values are recorded through a wireless sensor network deployed in the testbed and stored in a database. The temperature values are measured in degrees Fahrenheit, and the damper positions take one of two values: 0 (closed) or 1 (open 100%). The system status allows the model to see whether the system is in off, heat/cool 1, or heat/cool 2 mode. An example of the damper, temperature, and system status for one room is shown in Figure 6.2.

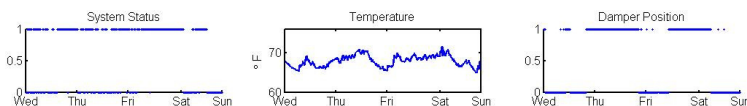


Figure 6.2: The system status (on/off), temperature ($^{\circ}\text{F}$), and damper position (open/closed) for one room in the testbed over the period 11/30/2011-12/04/2011.

In analyzing this system, a number of different models are explored. Three iterations of the final model are shown in the following sections. Each is a dynamic, linear model that is developed in two stages. The first stage aims to estimate the effects of heating/cooling due to external factors such as solar radiation, wind, and cloud coverage. This effect is calculated when the system is turned off, and the values are then used to develop the model when the system turns back on. This two-stage approach allows the compensation for external factors without having to measure them

directly. Furthermore, the results allow the prediction of temperature dynamics due to the HVAC configuration with greater accuracy.

$$dT_k/dt = \alpha T + \beta D \quad (6.1)$$

The models discussed follow the same format (Equation 6.1) in which the temperature of a specific room T_k over time t is a result of external factors (calculated through α), and the current damper configuration, D . The three iterations of this model differ in the way that the external factor coefficient, α , is calculated. These differences are as follows: 1) The first iteration calculates a universal α value by pooling the data when the system is off. 2) The second iteration calculates a constantly changing α value when the system is off, and uses this constantly calculated α value in the model when the system turns on. 3) The third iteration adds to the model complexity by using universal α values for all neighbors T_1, \dots, T_n of the temperature in room k , T_k .

Static α

The first iteration of the model described is one in which the α values, which estimate the temperature change due to weather patterns, are constant throughout the day. In order to calculate these values, the data from times when the system is off is pooled together and fit one α value across all timesteps for each of the n rooms. This value is calculated through linear regression, and assumes that the heat load due to weather remains relatively constant throughout the day.

Dynamic α

In the second iteration, the idea that the heat load due to weather conditions may be changing continuously throughout the day is explored. In order to do this, a dynamically changing α value is calculated for each off segment, and include that value in the on segment that directly follows it. This method aims to compensate for weather by assuming that the heat load due to weather changes significantly throughout the day, but by very little between one cycle of the system.

Adjacency Model

The third iteration increases the complexity of the first by including the other n rooms into the model. This assumes that the current temperature of the room is affected not only by its own

weather conditions, but also by the temperature dynamics within the other rooms of the building. This model also calculates the α values universally through linear regression. The form of this room adjacency model is as follows:

$$dT_k/dt = \alpha_1 T_1 + \alpha_2 T_2 + \cdots + \alpha_n T_n + \beta D \quad (6.2)$$

6.1.6 Analysis

Visually examining the measured and predicted temperatures (Figure 6.3) highlights a few modeling errors. One is that the predicted temperatures fail to predict periods of rapidly changing temperature. However, because the temperatures within this region are acting within a narrow range, this rapid change is likely to appear as noise to the system, and is reasonably captured by the predictive model.

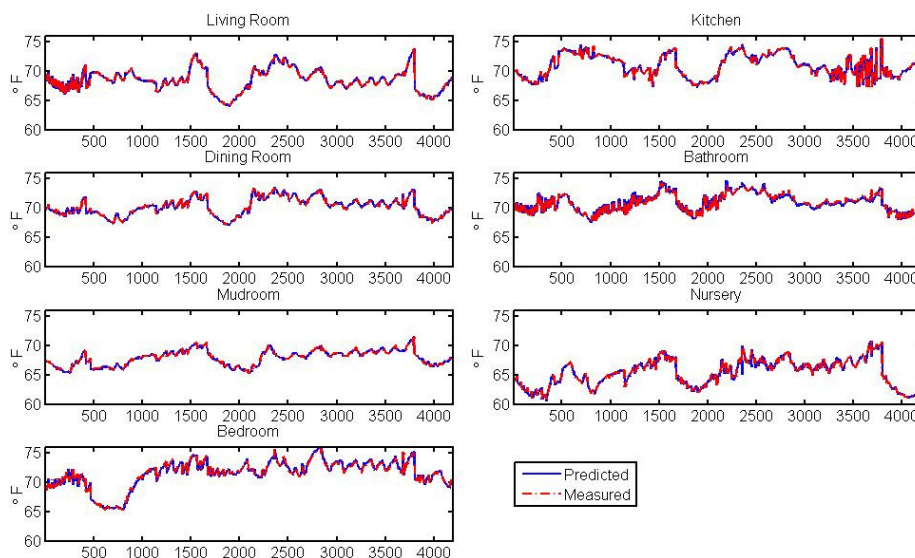


Figure 6.3: A plot of the predicted and measured temperatures when the HVAC system turns on.

Another visible error is that the predicted temperatures fail to capture the temperature magnitude in some cases throughout the week. The largest error is about 4 degrees Fahrenheit, a value which may be due in part to lost sensor data. Although this may appear large, it still lies within an interval that would ensure comfortable temperatures to occupants when creating a control scheme. Furthermore, the average error is .1064 °F, which is quite low and would certainly ensure comfort and precision within the system. Another indication of a good fit is that the root-mean-square error (Table 6.1) for this prediction is relatively low.

Table 6.1: Root Mean Square Errors

| Room | RMS Error ($^{\circ}\text{F}$) |
|-------------|----------------------------------|
| Living Room | 0.2037 |
| Kitchen | 0.3248 |
| Dining Room | 0.1228 |
| Bathroom | 0.3515 |
| Mudroom | 0.0847 |
| Nursery | 0.1858 |
| Bedroom | 0.2160 |

One difficulty in analyzing the effectiveness of this model is the goal of using it to predict temperatures more than one step in advance. This involves calculating predictions at each point that the system is on, up to t minutes into the future until the system turns off again. This analysis is visualized by showing the distribution (within two standard deviations) of the model errors aggregated over each prediction, x minutes into the future.

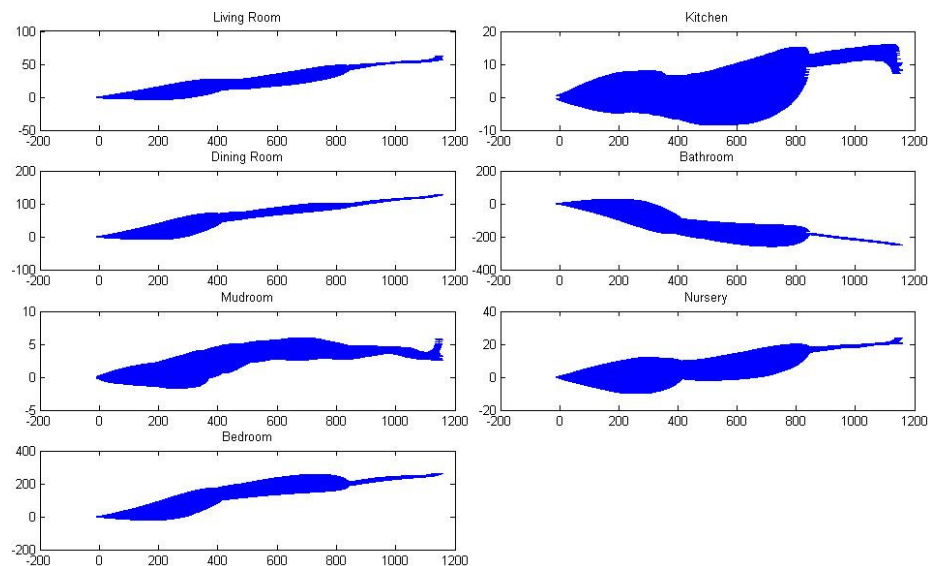
Figure 6.4: Aggregate plot of error distributions for each prediction, x minutes into the future.

Figure 6.4 shows that the error rates are much worse for some rooms than for others. These rates correspond directly to the β values calculated for these models. That is, the greater the β values for a given room, the worse that it may do in the extreme long-term prediction. This is because the predicted values grow linearly over time despite nonlinear temperature dynamics.

However, within a realm of reasonable prediction (less than 30 minutes), the predictions all do fairly well. As can be expected, the average errors tend to increase as predictions are made further

into the future. The variance of these errors also increases until a certain point, in which case the errors all become more uniformly skewed.

6.1.7 Results

Three iterations of the model described in section 5 are compared using 21 days worth of data tested with 10-fold cross validation which involves randomly dividing the 21 days of data into ten equal sets, training the model using nine of those sets, and testing with the remaining set. All combinations of nine sets for training and one set for testing are used. The 21 days selected for model development and testing have been sampled from 3 months worth of data between October and December 2011. Using the training data, the β values for the model are developed. These values are then used with the α value scheme dictated by the model iteration in order to predict temperatures when the system turns on.

Prediction

The predictions assume that temperature grows linearly when the system turns on as a result of the current damper configuration and the previous weather patterns estimated through α . Though temperature dynamics within a building are often nonlinear, a reasonable estimate is found by predicting temperature linearly into the future. This is because the temperature and airflow of the system operate within a narrow regime, making it reasonable to approximate change with a linear model. An example of a prediction 30 minutes into the future is shown in Figure 6.5. Here, the blue lines represent the actual temperatures and the red line plots the prediction. The solid blue line shows the temperature when the system is off, and the red/blue dashed lines show the predicted/actual temperatures when the system has just turned on.

Error Metric

The error metric chosen for the evaluation is to determine the distribution of prediction error as predictions are made t minutes into the future. For each minute, t , the mean and standard deviation of the prediction errors t minutes away from the initial time are calculated. The results from these analysis for the static α , dynamic α , and adjacency model are shown in Figure 6.6, Figure 6.7, and Figure 6.8 respectively. These results are calculated on a per-sensor basis for each of the 12 sensors in the 7 rooms of the building.

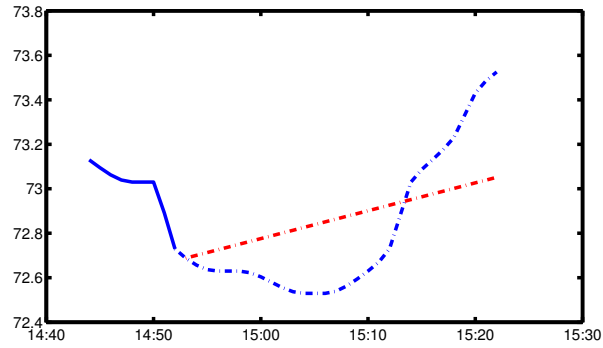


Figure 6.5: An example of a prediction made for temperature up to 30 minutes into the future after the system turns on. The solid blue line shows the actual temperature when the system is off; the dashed blue line shows the actual temperature when the system is on; and the dashed red line shows temperature predicted after the system has just turned on.

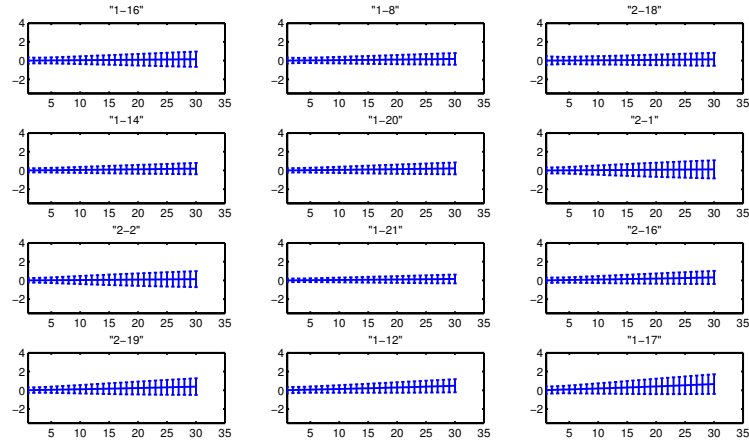


Figure 6.6: Error distributions for the static α model, up to 30 minutes into the future. The locations of the twelve sensors are presented in Figure 4.9

Visually examining the error distributions highlights a few important things about the model. The first observation is that variance of the errors tend to increase as predictions are made further into the future. The error can get quite large in some places, particularly in the dynamic α model. However, most of the values for each model remain within 2 degrees for the 30 minute prediction. This is a reasonable interval with which to enable the control of a predictive zoning system.

The results from this analysis also indicate that the simple, pooled α model performs better than the dynamic model. This may be counterintuitive since weather tends to change significantly

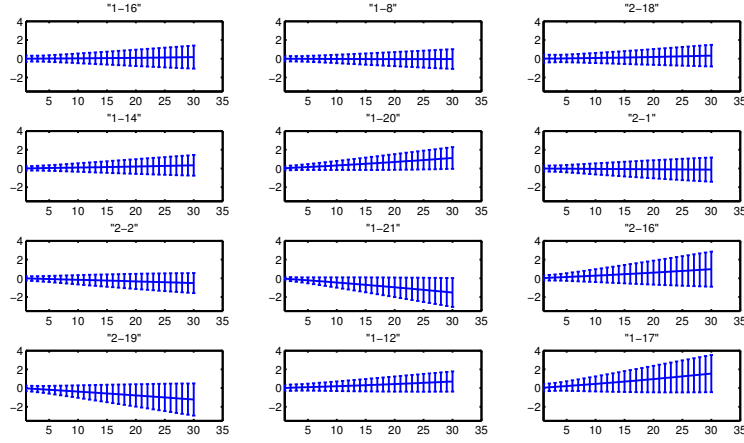


Figure 6.7: Error distributions for the dynamic α model, up to 30 minutes into the future. The locations of the twelve sensors are presented in Figure 4.9

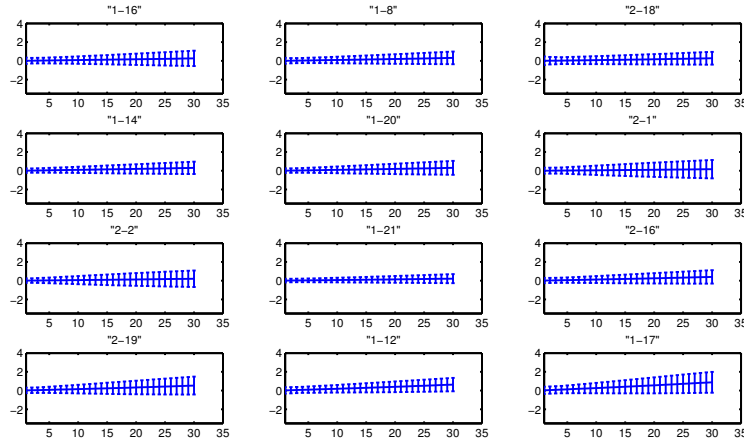


Figure 6.8: Error distributions for the adjacency model, up to 30 minutes into the future. The locations of the twelve sensors are presented in Figure 4.9

throughout the day. However, because of the window being looked at and the narrow range of temperature changes, it is reasonable that this model should perform well. It also has the added benefit of being computable and easy to implement within a control setting.

6.1.8 Conclusions

This section presented a model that can be used for thermal prediction in a zoned HVAC system. The two-stage, dynamic model presented provides an accurate way to predict the temperature in a zone based on a few, accessible parameters in the system. It also allows the calculation of highly variable terms, such as the heat load due to solar radiation, wind, and cloud coverage, without the need to explicitly measure these terms. This model can be used to implement the predictive zoning controller. The model provides better insight into the dynamics of the control scheme and allows for a more efficient design.

6.2 Predictive Occupancy Model

In order for room occupancy to be predicted, accurate models of occupancy are needed. This accuracy usually depends on the granularity of the model where a very *fine-grained model* that captures a large number of features is more accurate than a *coarse-grained model* composed of fewer features. There are many features, such as the current state of the room for which a prediction is required, the states of neighboring rooms, the time of day, etc., that can be considered when making occupancy predictions. Using all possible features to make a prediction is impractical due to the large amount of data necessary to train such a model. Thus, a fine-grained model would be unusable for a long period of time until sufficient data is collected for it to be useful. Making such a system usable in a home might require months, if not years, of training data collection. Thus, a trade-off has to be made between accuracy and training time. The traditional approach is to identify the features with the greatest predictive power and use only them to build an occupancy model. Yet, selecting these features is not straightforward due to the difficulty of separating noisy features from those that are highly correlated with occupancy. Furthermore, every house is different and, thus, a model composed of a single set of features would not prove accurate across all houses. They also vary within a home depending on the context. For instance, certain rooms such as the living room and kitchen are used very frequently and thus a model with many features can be built within a relatively short period of time, but a room such as the laundry room, may be used infrequently and thus a more general model, with fewer features, maybe necessary to be usable within a short period of time.

SmartZone uses a hierarchical model, called *Percolator*, to predict the room-level occupancy in

homes. In this section we describe the implementation of Percolator and evaluate its performance. Percolator can predict occupancy with accuracies sufficient for most smart home applications. A feature of Percolator is its ability to automatically increase in accuracy as more data is collected during the usage of the system. Using a hierarchy of models with various sets of features also has the added benefit of allowing models to be selected that match room usage patterns. This would allow fine-grained models to be built quickly for rooms that are frequently coarse-grained models can be trained for rooms that are infrequently used. These attributes of a hierarchical model makes Percolator ideal for the many smart home applications that require the occupancy of rooms to be predicted with a minimal training period.

What most smart home systems require is an occupancy predictor that is sufficiently accurate within a couple of weeks of the system being installed, but improves in its predictions using the data collected as the system operates over time. One approach to implement such a predictor is to define a model composed of a large number of features, but uses only a few of its features to make predictions initially. Over time, as more data is collected, additional features are activated. The problems that have to be solved to implement such a system is deciding when to activate another feature and the order in which the features should be activated.

Percolator addresses these problems by utilizing a hierarchical occupancy prediction approach that uses a set of models ranging from general models, with few features, to specific models with many features, and automatically selects a model to make the most accurate prediction possible for a particular situation. The main benefit to the Percolator approach is the ability for a smart home application to be able to predict occupancy with over 80% accuracy within ten days of being deployed with the prediction accuracy increasing the longer the system is deployed.

6.2.1 Background and Related Work

The Neurothermostat [130] uses a hybrid occupancy predictor that leverages a daily schedule and a neural network. Yet, it needed to be trained on five consecutive months of data to predict occupancy accurately. The reliance of this approach on a daily schedule and almost half a year's worth of occupancy data make it infeasible for most smart home applications. Krumm et al. [154] present an occupancy prediction algorithm that gives the probability of occupancy at different times for each day of the week. This approach would work well for households where the residents have very

similar occupancy patterns for a particular day of the week, but is unable to respond to changes in occupancy patterns for a particular day.

The DGQL Occupancy Prediction Model (DOPM) is an occupancy prediction model built using the Decision Guidance Query Language (DGQL) [155]. Its aims are to maximize energy saved in a location while limiting the inconvenience caused to its occupants. DOPM is based on presenting prediction rules, which it then refines using motion sensor data by maximizing the total empty time and attempting to limit the number of errors to fewer than n where n is an acceptable inconvenience. A shortcoming of DOPM is that it relies heavily on the prediction rules. Therefore, providing it with insufficient, or incorrect, rules could result in inaccurate predictions.

The Smart Thermostat [128] uses a Hidden Markov Model (HMM) to predict occupancy at the house-level. This approach was not evaluated for room-level occupancy prediction but would necessitate a large amount of data to train an HMM per room. The HMM approach is similar to other approaches that use Bayesian probability theory to predict occupancy. Page et al. [156] use a generalized stochastic model to simulate occupancy presence as an input for future occupant behavior models within building simulation frameworks. Dodier et al. [157] use sensor belief networks in order to detect occupancy. The belief network was used to generate a probability distribution of occupants and their locations over time given historical sensor data. Yet, this approach also suffers from requiring a large amount of training data to provide useful predictions for smarthome applications.

The latest approach at using occupancy prediction for thermostatic control is by PreHeat [158]. Occupancy is detected using RFID tags placed on house keys to sense if the house is occupied and motion sensors in each room to sense room occupancy. Occupancy is recorded as a binary vector for each day where each element of the vector represents the occupancy in a 15 minute interval with 1 indicating the room was occupied and 0 indicating the room was vacant. PreHeat predicts the next time a room will be occupied by searching for the five closest matching days to the current day based on hamming distance and then calculating the mean of the occupancy values for these days. PreHeat was implemented in three homes in the United States and two homes in the United Kingdom. Leveraging the room-level radiators and underfloor heating units commonly used in the UK, PreHeat provides room-level heating control. The implementation of PreHeat in the United States resorted to controlling the HVAC system based on whole house occupancy due to HVAC systems in the US traditionally being centralized units. PreHeat works well for homes

with regular occupancy patterns, but would not work as well for homes where the occupants do not have regular schedules. The method also requires a large amount of historical data in order to find five days with occupancy patterns closely matching any given day. The main weakness of PreHeat that Percolator attempts to address is the lack of interaction between rooms in any of the models used for prediction. For instance, the authors predict the occupancy of each room based on the history of occupancy of that particular room without considering the occupancy of any other rooms. Taking into consideration occupancy patterns across a house would lead to higher accuracies in predicting the occupancy of a particular room. The authors also use a very simple thermal model to predict when a room should be preheated. The model is simply the average amount of time it took to increase the room temperature by a degree based on historical data. Predictive zoning uses a model that takes into consideration the thermal interactions between rooms. Also, PreHeat does not provide a room-level solution for houses with centralized HVAC systems which RoomZoner and Smart Zone provide.

6.2.2 Approach

Percolator is composed of a set of *predictors*: occupancy models built from historical occupancy data showing how many times a particular set of features has been observed in the past, the *total observation count*, and how many times a room has been occupied when these features were observed, the *room occupancy count*. These counts are for various *prediction horizons*: times into the future for which predictions are desired. In the implementation presented in this section, prediction horizons of 1 minute, 2 minutes, 3 minutes, 5 minutes, 10 minutes, 20 minutes, and 30 minutes are used. Table 6.2 shows an example of a simple predictor that uses only the current time as the feature.

Table 6.2: Example of a predictor with current time as feature.

| Time | Room 1 @ 1 min | Room 1 @ 2 min | ... | Room N @ 30 min | Total Observations |
|-------|----------------|----------------|-----|-----------------|--------------------|
| 00:00 | 5 | 0 | 15 | 1 | 30 |
| 00:01 | 4 | 1 | 20 | 5 | 31 |
| 00:02 | 1 | 3 | 22 | 0 | 30 |
| ... | ... | ... | ... | ... | ... |

In Percolator the predictors are composed in a hierarchical manner ordered from the most fine-grained predictor to the most coarse-grained predictor. Figure 6.9 illustrates an instance of Percolator with a *percolation depth*, the number of predictors composing the model, of two. As the arrows indicate, Percolator is ordered from fine-grained predictors to coarse-grained predictors. Also, a

smaller percentage of all possible feature combinations would have been observed for a fine-grained model than a coarse-grained model because there are more possible combinations for a fine-grained model.

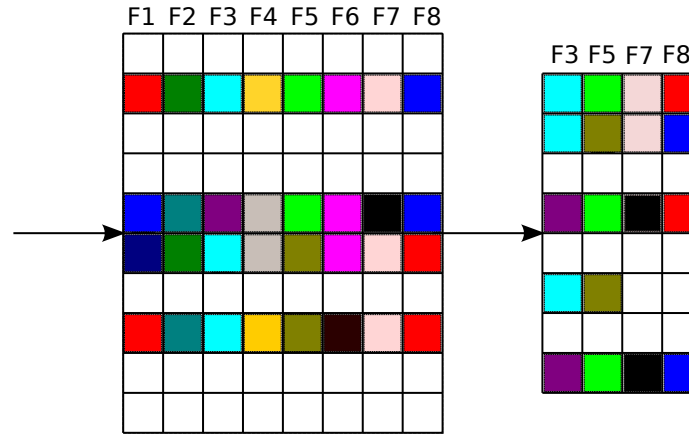


Figure 6.9: An illustration showing an instance of Percolator composed of two predictors. The first predictor is composed of eight features ($F1 - F8$) while the second predictor is composed of a subset of four of these eight features. Filled rows indicate a particular combination of features that has been observed while empty rows indicate feature combinations that have not yet been observed. The colors illustrate the values that the feature could take. For instance, if the feature is room state, the colors indicate occupied or vacant. This figure only illustrates feature combinations. In Percolator, each feature combination has an associated room occupancy count and total observation count as shown in Table 6.2.

Figure 6.10 illustrates an observation of the eight features for which an occupancy prediction in the future is desired. There is no row in the fine-grained predictor that matches this combination of features. Therefore, Percolator percolates the prediction decision to the next predictor in the hierarchy. In this instance, features 3, 5, 7, and 8 that are observed match the second row of the coarse-grained predictor and therefore the room occupancy count, C_R , for the room and prediction horizon, k , of interest and total observation count, C_T , associated with this set of features is obtained from this predictor. The probability of occupancy, P_O , is calculated as $P_O = C_R/C_T$. If this fraction is greater than an *occupancy threshold* the room is predicted to be occupied k minutes in the future.

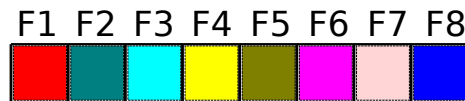


Figure 6.10: An illustration of a set of observed features. These observed features can be used to make a prediction of future occupancy using Percolator.

Making a prediction based on a few prior observations could result in incorrect predictions being made. In order to minimize such occurrences, Percolator uses a *percolation threshold*: the minimum number of observations on which a prediction can be made. If the total observation count is below the percolation threshold, the prediction decision is percolated up the hierarchy. As more observations are expected with coarser-grained predictors, percolating up the hierarchy increases the probability of using a predictor that satisfies the percolation threshold.

In the event that none of the predictors in the hierarchy has sufficient historical data to make a prediction, Percolator has a number of fall-backs it can use to make a prediction. The most simple of these fallback techniques is *current state* where the model simply predicts that the room would simply remain at its current state, whether occupied or unoccupied. Another fallback technique evaluated is *hamming distance* where, in the event Percolator cannot find sufficient historical days even with the most general model, it looks for the day with the set of features closest to the currently observed feature set based on hamming distance and use the observed occupancy of the room at that day as the prediction. This method proved too computationally intense to be practical, and therefore *current state* was selected as the fallback for Percolator.

6.2.3 Experimental Setup

| #Residents | #Rooms | #Motion Sensors | #Weeks |
|------------|--------|-----------------|--------|
| 2 | 7 | 14 | 12 |
| 1 | 7 | 8 | 24 |
| 2 | 10 | 10 | 8 |
| 3 | 7 | 7 | 12 |

Table 6.3: Details of the 4 homes from which occupancy data was collected

Data collected from four residences over periods ranging from two to six months was used to evaluate Percolator. These deployments included both single- and multi-person homes with varying occupancy patterns. The residents ranged from students and working professionals to stay at home parents and residents who worked at home. Table 6.3 summarizes the information about the homes. Ten-fold cross-validation is used to train and test the model using the collected occupancy data to generate the results presented in Section 6.2.4. Figure 6.11 shows the prediction hierarchy used for the evaluation.

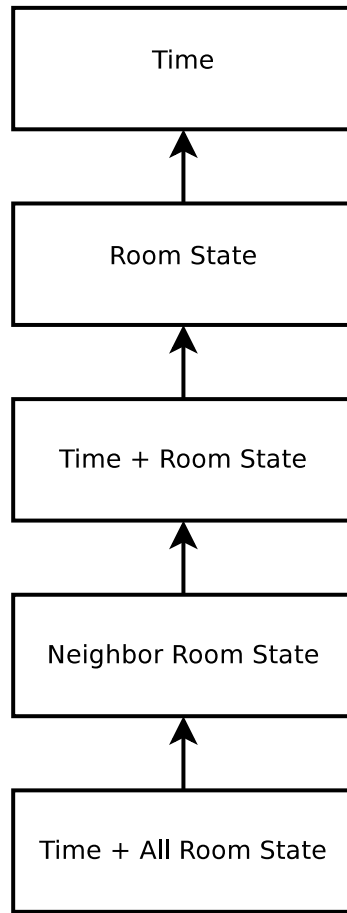


Figure 6.11: The Percolator hierarchical occupancy predictor starts at the bottom of the hierarchy with the most specific, and accurate, occupancy model and percolates up towards more general models until a perfect match of features is found.

6.2.4 Results

The performance of Percolator is evaluated across the four houses as the amount of training data available increases. As Figure 6.12 shows, Percolator achieves over 75% accuracy within 10 days of deployment with accuracy increasing as it is trained with more data.

6.2.5 Sensitivity Analysis

Sensitivity analyses are performed to understand how varying the parameters associated with Percolator affects its performance.

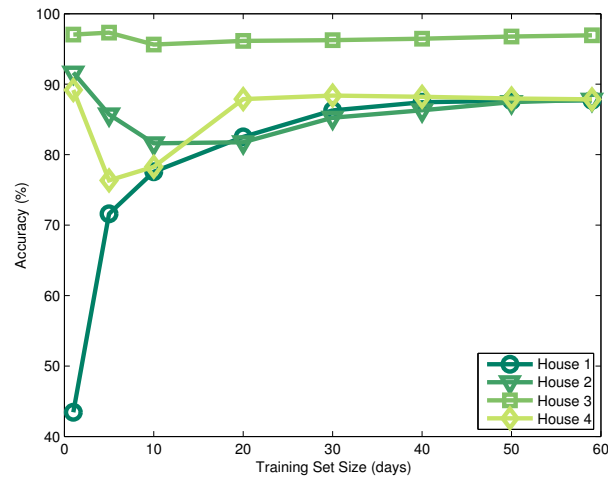


Figure 6.12: Percolator increases in accuracy with more training data achieving over 75% accuracy across all four houses with only 10 days worth of data.

Training Set Size

Figure 6.13 shows the effect on the accuracy as Percolator is trained with more data. Maintaining the sensitivity threshold at 3 and increasing the size of the training set from 10 days worth of data to 60 days of data results in an accuracy increase from 74% to 83%. This increase in accuracy is due to more decisions being made with the more accurate fine-grained predictors as more data is available to train the model. At most training set sizes Percolator outperforms the component predictors by efficiently selecting between them to make an accurate prediction. Also, as expected the accuracy of Percolator increases as more data is available for training.

Figure 6.14 also shows that the predictor with the greatest accuracy varies from house to house and as more data is collected. For instance, in House 1 with 20 days of training data the most accurate predictor is the one that uses the current time and states of all the rooms as the prediction features while for House 2, with the same amount of training data, the predictor that uses only the neighboring room states is the most accurate. Also, with less training data coarse-grained predictors such as those that use only the state of the room of interest is more accurate than finer-grained predictors, but as more data is collected the finer-grained predictors become more accurate than the coarse-grained predictors.

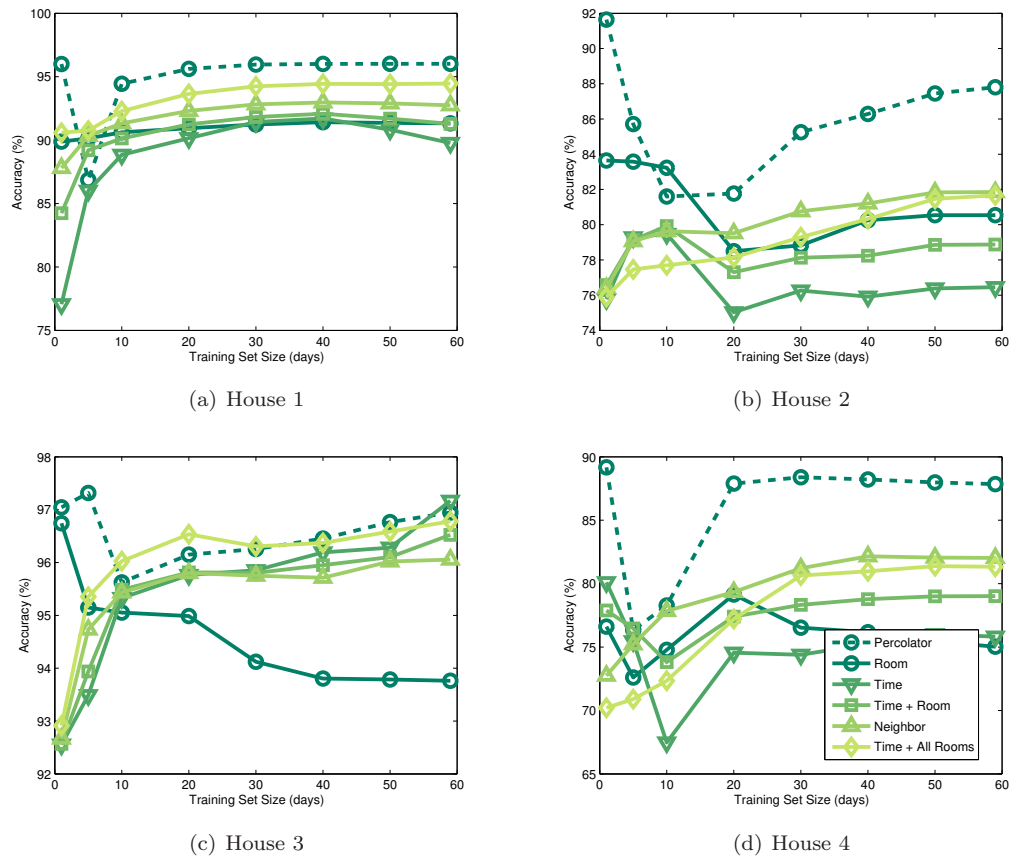


Figure 6.13: Percolator achieves over 65% accuracy in predicting rooms to be occupied in all four houses with two months of training data. Three of the four houses achieve over 75% accuracy with only 20 days of training data.

Prediction Horizon

As with most models that attempt to make predictions, Percolator gets less accurate as it makes predictions further into the future from the current time. Figure 6.15 shows the effect on the accuracy across the four houses as this *prediction horizon* increases from one minute in the future to 30 minutes into the future. With only 60 days of training data, occupancy could be predicted with about 60% accuracy 20 minutes into the future and for shorter prediction horizons over 90% accuracy is possible. This accuracy is expected to increase as more training data is available.

Figure 6.16 shows the effect of increasing the prediction horizon on the five predictors that compose Percolator. Percolator does better than all predictors until a prediction horizon of 10 minutes. Beyond that horizon, it does as well as the neighbor room state predictor. More sophisticated algorithms for percolation thresholding are being developed to transition between predictors and select

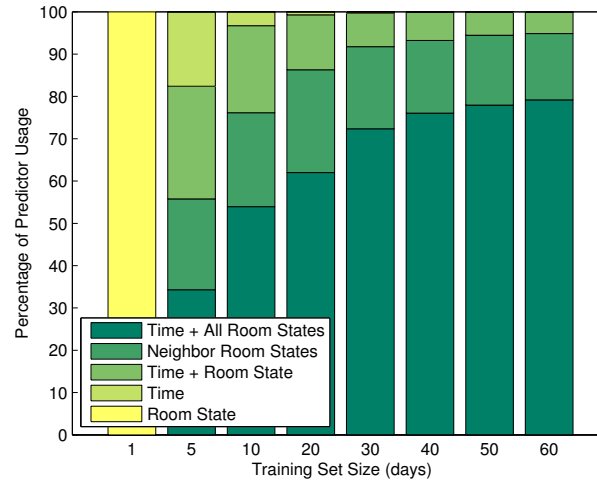


Figure 6.14: As the training set size increases, a greater fraction of the decisions are made using the more specific predictors.

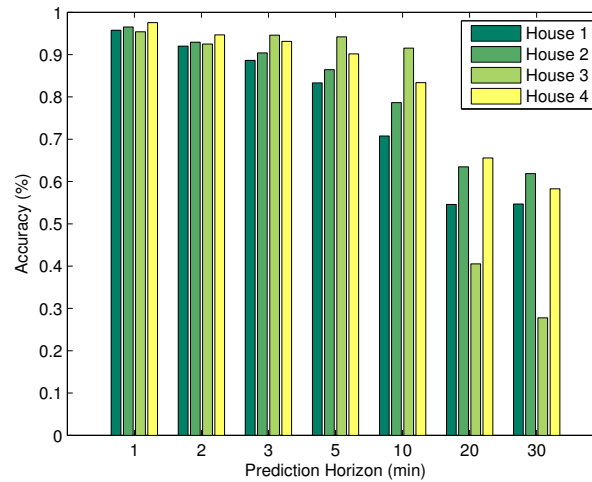


Figure 6.15: The average accuracy of predictions across the four house drops from 96% to 51% as the prediction horizon increases from one minute to 30 minutes.

better predictors at further prediction horizons. This would enable Percolator to do at least as well as the most accurate predictor at any prediction distance.

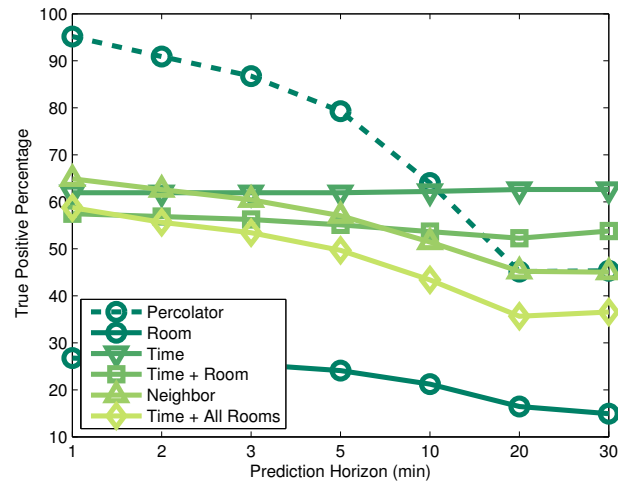


Figure 6.16: For the first house, Percolator does better than all component predictors for short prediction horizons and performs as well as the predictor with the average accuracy for horizons further into the future. The solid lines show the five predictors that compose Percolator, and the dashed line shows the accuracy of Percolator.

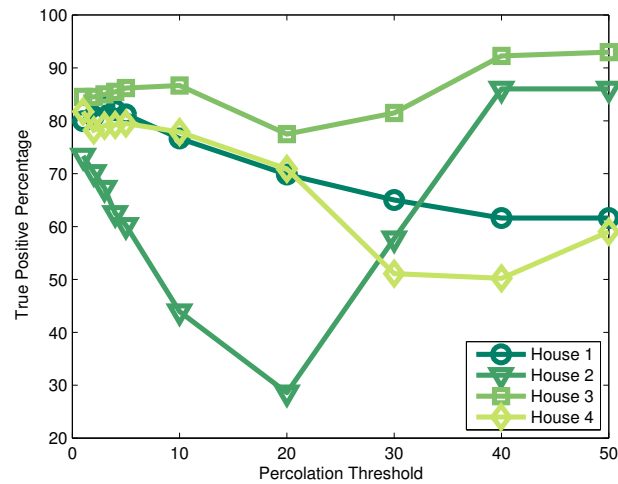


Figure 6.17: Prediction accuracy increases as the percolation threshold increases until the threshold is too high for the amount of data used to train the model.

Percolation Threshold

As described in Section 6.2.2 the percolation threshold is the minimum number of days with identical features necessary before an occupancy predictor can be used make a prediction. If the threshold

is not met, a less specific predictor is tried. Figure 6.17 shows the effect of varying the percolation threshold on true positive accuracy for one of the houses. The model was trained with 60 days worth of data and tested on another day's worth of data. The graph shows that as the percolation threshold increases from 1 to 3, the prediction accuracy increases. This is due to the fact that using more historical data to make a prediction would result in a more accurate prediction. Yet, a further increase of the percolation threshold, from 3 to 5, causes a drop in accuracy. This is due to 60 days of training data being insufficient for four or five identical instances to be found for the more accurate specific predictors, and therefore the decision would percolate up the hierarchy resulting in a less accurate, more general predictor being used to make the prediction as shown in Figure 6.18.

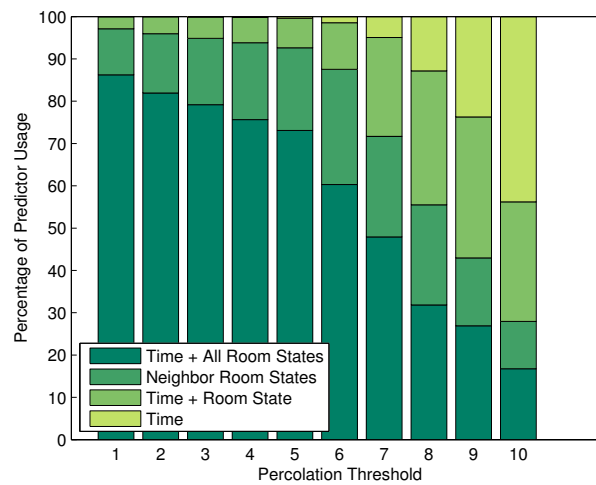


Figure 6.18: As the percolation threshold increases, more decisions percolate up the hierarchy from specific models towards general models.

Percolation Depth

Percolation depth is the depth of the percolation hierarchy. This is varied by changing the number of predictors composing Percolator. Figure 6.19 shows that adding more models to Percolator increases its accuracy across all training set sizes. At three models and 20 days worth of training data Percolator is as accurate as it is with four or five models.

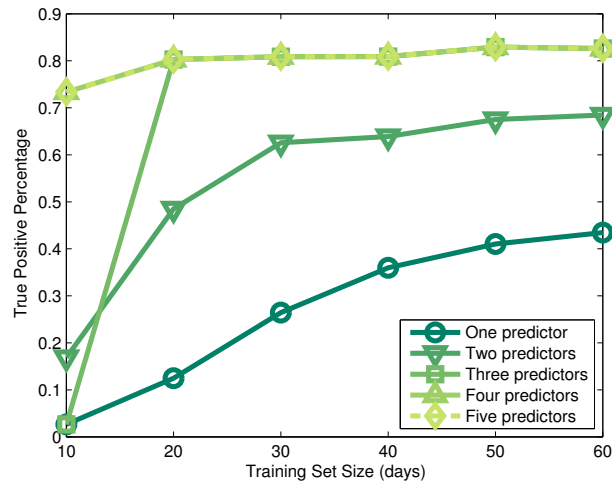


Figure 6.19: As more predictors are added to Percolator, the accuracy increases.

6.2.6 Conclusions

This section presents Percolator, a model that can be used for occupancy prediction in a zoned HVAC system. Percolator addresses the issue of accurate occupancy prediction requiring large amounts of data on which to train models, yet most applications requiring accurate occupancy prediction within weeks of being deployed. Percolator exploits a hierarchy of occupancy predictors of decreasing granularity to enable occupancy predictions with over 75% accuracy to be achieved within 20 days of a system being deployed. An added benefit of Percolator is its ability to improve its prediction accuracy as more data is collected. Thus, over time as the smart home application continues to be used predictions with greater accuracy would be possible.

6.3 Predictive Control

The predictive thermal model and the predictive occupancy model can be used to implement Smart-Zone. One approach for such a predictive zoning controller would be to optimize over a search space of all possible actuation decisions attempting to minimize the following cost function:

$$C = \alpha E + \beta M \quad (6.3)$$

where E is the expected energy expenditure for the control decision and M is a *miss penalty* that

captures the discomfort a resident could expect due to the control decision. E can be calculated as

$$E = e_s * t \quad (6.4)$$

where e_s is the energy consumed by a particular HVAC stage and t is the amount of time for which the stage is expected to be active. Percolator and the predictive occupancy model are used to estimate M . Percolator is used to predict which rooms are occupied from the current time until the next control decision will be made and the predictive thermal model is used to estimate the effect of a particular damper configuration and HVAC stage on the temperatures of those rooms. The total amount of time the rooms are above or below the setpoint is M . α and β are weighting functions that can be exposed to the residents as knobs in order for them to be able to trade-off comfort for energy savings. By increasing α relative to β the residents could indicate that they are willing to tolerate a longer lag-time in order to save energy while increasing β relative to α would indicate the residents prioritize their comfort over potential energy savings.

6.4 Macroprogramming Discussion

While SmartZone has not yet been implemented, two of its main components the predictive temperature and occupancy models were implemented using Matlab and Python respectively. The Matlab-based occupancy prediction code can be added to a MacroLab implementation of SmartZone with very minimal, if any, alterations due to MacroLab supporting all native Matlab functions. The Python implementation of Percolator heavily relied on the Numpy package which provides Matlab-like functionality within Python. Therefore, porting this code to MacroLab would require very little effort. Figure 6.20 shows the high-level logic implemented in MacroLab.

The trend of increasing application complexity decreasing the freedom with which MacroLab can optimize code implementation continues with SmartZone. While RoomZoner could benefit from in-network aggregation in larger homes with multi-hop networks, SmartZone cannot aggregate any sensor values due to all sensor data being required to train, and improve, the predictive models. Macroprogramming languages are still beneficial to implementing such applications due to the simpler programming abstraction they provide over node-level programming.

```

1 RTS = RunTimeSystem();
2 weatherdirect = RTS.getMotes('type', 'weatherdirect');
3 tempSensors = SensorVector(weatherdirect, 'temperature');
4 x10 = RTS.getMotes('type', 'X10');
5 motionSensors = SensorVector(x10, 'motion');
6 motionSensorIDs = uint8([8 1 9], [2 6], [10 5], [4], [7 11], [12 14], [3 15]);
7 tempSensorIDs = uint8([1 3], [2], [6 7], [4 9 11], [12 13], [5 14], [8 10]);
8 damperIDs = uint8(1:15);
9 nightStart = [0 0 0 2 0 0];
10 nightEnd = [0 0 0 7 0 0];
11 curState = 'On'
12 every(60000)
13     mode = dbRead('zoning', 'mode', 'latest')
14     motionVals = motionSensors.sense();
15     tempVals = tempSensors.sense();
16     occupancyPrediction = Percolator(motionVals, 1)
17     hvacChoices = []
18     damperConfigs = []
19     costs = []
20     for hvacStage = 0:2
21         for numClosedDampers = 0:15
22             for damperCombination = combntns(damperIDs, numClosedDampers)
23                 temperaturePrediction = tempPredictor(hvacStage, damperCombination,
24                     tempVals)
25                 cost = costEstimator(hvacStage, damperCombination, occupancyPrediction,
26                     temperaturePrediction)
27                 hvacChoices(end + 1) = hvacStage
28                 damperConfigs(end + 1) = damperCombination
29                 costs(end + 1) = cost
30             end
31         end
32     end
33     minIndex = find(costs == min(costs))
34     optimumStage = hvacChoices(minIndex)
35     optimumDamperConfig = damperConfigs(minIndex)
36     hvacActuate(mode, optimumStage, optimumDamperConfig)
37 end

```

Figure 6.20: MacroLab implementation of SmartZone.

6.5 Conclusions

This case study demonstrates the benefit of an easy to use high-level programming model to implement a complex CPS application. SmartZone involves all aspects of the most sophisticated CPS applications: data collection, real-time computation, actuation, and prediction. Due to all the collected data being required to train the prediction models, a macroprogramming abstraction does not have any flexibility to optimize the code implementation. Yet MacroLab greatly eases the application

development process by presenting the programmer an abstraction of all the data as vectors. The programmer is not burdened with low-level network details such as sending and receiving messages or handling dropped packets and instead can focus on writing, and debugging, the application logic which in itself is a difficult task for such a complicated CPS application. While SmartZone was not implemented due to such a study requiring another year of data collection, the major components for its implementation, the predictive temperature and occupancy models, are evaluated in this chapter. An overview of an approach to use these models to implement SmartZone is also presented.

Chapter 7

Conclusion

In this dissertation, we present an implementation of a macroprogramming and macrodebugging framework for Cyber-Physical Systems and discuss an application of CPSs in the form of a room-level zoning system to minimize the energy consumed for heating homes by conditioning only occupied spaces. Macroprogramming systems make programming CPSs easier by providing high-level distributed abstractions such as database tables, logical facts, or data streams so that users do not need to build mental models of the individual nodes and their interactions. With MacroLab we approached the issue of software design for CPSs by attempting to address the central question of whether programs should be implemented in a centralized or distributed fashion. Most early sensor network research focused on “*localized* algorithms – where sensors only interact with other sensors in a restricted vicinity, but nevertheless collectively achieve a desired global objective” [50]. For example, early object tracking applications argue that neighborhood communication and local processing are necessary to efficiently filter false positives [12] and services like TAG use *in-network aggregation* to calculate network statistics *en route* to decrease message passing [4]. More recently, several architectures have proposed the use of centralized algorithms to control distributed systems. *Marionette* allows the user to write a centralized Python script to control all nodes in a network [18]. It argues that centralized algorithms are easier to write and debug and that, once debugged, functionality can be *migrated* to the sensor nodes for efficiency reasons if necessary [7]. The *Tenet* [22] architecture takes a stronger stance by arguing that all application-specific code should *always* execute on master devices while sensor nodes should be restricted to a small set of predefined operations on locally-generated data. The rationale here is to separate the application code from the networking code so

that changes in the application do not cause cascading changes to the networking middleware.

With the *deployment-specific code decomposition* approach taken by MacroLab, we argue that programs can actually be implemented in both a centralized *and* a distributed fashion. We re-frame the architectural question from *where code should execute* to *how code should be written*. The central tenet of our architecture is that all application-specific logic should be contained in a macroprogram and all distributed operations must be contained as libraries in the run-time system. When code is written in this way, we get the best of both worlds: (1) the decomposer and the run-time system can choose the manner of implementation that provides the best performance in terms of cost metrics like bandwidth, power, and latency, and (2) the user is free to write *deployment-independent* programs that are simple, robust, and easy to understand.

While there has been considerable research in the area of macroprogramming abstractions for CPSs, to the best of our knowledge, no macroprogramming systems have debugging support, which is a crucial link in the development chain as a macroprogram progresses from the drawing board to real deployment. We present MDB, the first system that allows the user to inspect and analyze the execution of a WEN using the high-level abstractions provided by a macroprogramming system. This process that we call *macrodebugging* simplifies the debugging process and eliminates the need to analyze traces of low-level events and message passing algorithms. We show that macrodebugging is not only easier, but also more efficient than the debugging of node-level programs.

While the MDB macrodebugger is built for the MacroLab macroprogramming abstraction, the underlying principles can be applied to other macroprogramming systems. The collection of data traces can be applied to any system that has a high-level abstraction for which the overhead of logging a single high-level operation is small compared to the number of low-level instructions required to execute that operation. This property holds for most macroprogramming systems. The source-level debugging interface can be applied to any system that uses a sequential imperative language and has a clear mapping between macrocode and microcode, such as Kairos [20], Plaeiades [19], and Marionette [18]. Other systems that use functional [15] or declarative abstractions [4] must present information from data traces using a different interface. The four hypothetical changes presented illustrate the affects of adding data synchronization to a macroprogram, and are only useful to systems like MacroLab and Hood [12] that make heavy use of data caching. However, the general concept of creating hypothetical changes to illustrate how theoretical changes to a program or execution state *would* affect global state could be applied to aspects of other systems besides data

synchronization.

The current implementation of MDB provides *post-mortem* debugging, which means that it collects data about the system at run time and allows the user to inspect program execution after the logs are retrieved. At least some of the underlying concepts, however, could be applied to on-line debugging. For example, the process of logging data instead of events to recreate system state would reduce the amount of data that needs to be collected during on-line debugging, in the same way that it reduces data collection requirements for off-line debugging. Similarly, the logically-synchronous and temporally-synchronous source-level debugging interface could also be used for on-line debugging, and could even be combined with off-line data trace analysis to provide both forward and backward stepping. Hypothetical changes would not be as useful for on-line debugging as they are for off-line debugging, because the user could test changes to the system by simply changing the current state of the system before allowing execution to proceed.

Finally, we analyze the effectiveness of MacroLab as a CPS programming abstraction using a series of case studies based on zoning a centralized HVAC system in response to occupancy. As the case-studies demonstrate, as applications get more complex the amount of in-network and node-level data processing and code optimization decreases. In such a scenario, providing a clean simple abstraction which allows a programmer to write the program logic without worrying about where the data resides and how the data would be collected would greatly reduce the burden of CPS application development. The automatic optimization of applications based on topology and available hardware allows a programmer to be confident that a program written in a macroprogramming language such as MacroLab will execute as efficiently as one hand-tuned with node-level code.

As the current state-of-the-art and this dissertation demonstrates a large amount of work is necessary in order to make the available macroprogramming languages usable for a wide variety of applications. Programmers instinctively resort to the decade-old TinyOS programming environment simply due to the fact that it has the most complete set of libraries and tool-chain. The benefit of not having to implement any drivers of library functions in TinyOS outweighs the burden of programming at the node-level. The main reason for TinyOS being such a complete programming framework is it being the first widely adopted programming environment for wireless sensor networks, the precursor to CPSs. Due to not having any alternatives, pioneering application developers wrote the functions and tools they required for their applications and added it to the open-source TinyOS libraries. Over time sufficient functions and tools were added for later application developers to be

able to use them without having to write them from scratch.

A similar effort has to happen in the macroprogramming arena. Macroprogramming languages should be open sourced and made easily accessible with clear documentation on how to extend them by adding to their function libraries and tool-chains. Then, CPS application developers should be willing to sacrifice some time writing the drivers and functions they require for their applications when using the macroprogramming language. Achieving a complete macroprogramming environment through such a collaborative effort would greatly aid in the implementation of CPS applications and accelerate our advancement towards the world covered by sensors that was envisioned by the wireless sensor network pioneers over a decade ago.

Bibliography

- [1] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM.
- [2] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, August 2005.
- [3] A. Dunkels, B. Gronvall, and T. Voigt. Contiki-a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Washington, DC, USA, November 2004. IEEE Computer Society.
- [4] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.
- [5] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *NSDI '04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.
- [6] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph M. Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The design and implementation of a declarative sensor network system. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 175–188, New York, NY, USA, 2007. ACM.
- [7] Whitehouse: Issue energy & environmental.
http://www.whitehouse.gov/issues/energy_and_environment/.
- [8] The GDB developers. GDB: the GNU project debugger. <http://sourceware.org/gdb>.
- [9] The Mathworks. Matlab - the language of technical computing.
<http://www.mathworks.com/products/matlab/>.
- [10] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An Operating System for Sensor Networks. *Ambient Intelligence*, pages 115–148, 2005.
- [11] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, 2002.

- [12] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 99–110, New York, NY, USA, 2004. ACM.
- [13] J. Bachrach and J. Beal. Programming a Sensor Network as an Amorphous Medium. Technical report, MIT, 2006.
- [14] Geoffrey Mainland, Matt Welsh, and Greg Morrisett. Flask: A Language for Data-Driven Sensor Network Programs. Technical Report TR-13-06, Harvard University, May 2006.
- [15] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 489–498, New York, NY, USA, 2007. ACM.
- [16] U. Bischoff and G. Kortuem. RuleCaster: A Macroprogramming System for Sensor Networks. *OOPSLA '06: Proceedings OOPSLA Workshop on Building Software for Sensor Networks*, 2006.
- [17] D.C. Chu, L. Popa, A. Tavakoli, J.M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The Design and Implementation of A Declarative Sensor Network System. *SenSys*, 2006.
- [18] Kamin Whitehouse, Gilman Tolle, Jay Taneja, Cory Sharp, Sukun Kim, Jaein Jeong, Jonathan Hui, Prabal Dutta, and David Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks*, pages 416–423, New York, NY, USA, 2006. ACM.
- [19] Nupur Kothari, Ramakrishna Gummadi, Todd D. Millstein, and Ramesh Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 200–210, New York, NY, USA, 2007. ACM.
- [20] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming wireless sensor networks using Kairos. In *International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 126–140, 2005.
- [21] A. Awan, S. Jagannathan, and A. Grama. Macroprogramming heterogeneous sensor networks using cosmos. *SIGOPS '07*.
- [22] Omprakash Gnawali, Ki-Young Jang, Jeongyeup Paek, Marcos Vieira, Ramesh Govindan, Ben Greenstein, August Joki, Deborah Estrin, and Eddie Kohler. The tenet architecture for tiered sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 153–166, New York, NY, USA, 2006. ACM.
- [23] P.A. Vicaire, E. Hoque, Z. Xie, and J.A. Stankovic. Bundle: A group based programming abstraction for cyber physical systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, pages 32–41. ACM, 2010.
- [24] H. Richardson. High performance fortran: history, overview and current developments. Technical report, Thinking Machines Corporation, 1996.
- [25] Seema Hiranandani, Ken Kennedy, Charles Koelbel, Ulrich Kremer, and Chau-Wen Tseng. An overview of the Fortran D programming system. In *LCPC: Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 18–34, London, UK, 1992. Springer-Verlag.

- [26] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten Von Eicken, and Katherine Yelick. Parallel programming in split-c. In *Proceedings of Supercomputing '93*, pages 262–273, 1993.
- [27] Hongzhou Liu, Tom Roeder, Kevin Walsh, Rimón Barr, and Emin Gün Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 149–162, New York, NY, USA, 2005. ACM.
- [28] Galen C. Hunt and Michael L. Scott. The Coign automatic distributed partitioning system. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 187–200, Berkeley, CA, USA, 1999. USENIX Association.
- [29] Eli Tilevich and Yannis Smaragdakis. J-orchestra: Automatic java application partitioning. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 178–204, London, UK, 2002. Springer-Verlag.
- [30] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. Automatic data structure selection in setl. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 197–210, New York, NY, USA, 1979. ACM.
- [31] T. He, J.A. Stankovic, R. Stoleru, Y. Gu, and Y. Wu. Essentia: Architecting wireless sensor networks asymmetrically. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 1184–1192. IEEE, 2008.
- [32] Milan Jovanovic and Veljko Milutinovic. An overview of reflective memory systems. *IEEE Concurrency*, 7(2):56–64, 1999.
- [33] Rong-Guey Chang, Tyng-Ruey Chuang, and Jenq-Kuen Lee. Compiler optimizations for parallel sparse programs with array intrinsics of fortran 90. In *ICPP '99: Proceedings of the 1999 International Conference on Parallel Processing*, page 103, Washington, DC, USA, 1999. IEEE Computer Society.
- [34] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, December 2002.
- [35] G. Ramalingam. Data flow frequency analysis. In *PLDI '96*.
- [36] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, New York, NY, USA, 2004. ACM.
- [37] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. *IPSN '05*.
- [38] Crossbow. Micaz datasheet. http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAZ_Datasheet.pdf.
- [39] The mathworks. <http://www.mathworks.com/>.
- [40] C. Sharp, S. Schaffert, A. Woo, N. Sastry, C. Karlof, S. Sastry, and D. Culler. Design and implementation of a sensor network system for vehicle tracking and autonomous interception. In *Wireless Sensor Networks, 2005. Proceedings of the Second European Workshop on*, pages 93–107, 2005.

- [41] René Müller, Gustavo Alonso, and Donald Kossmann. A virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 41(3):145–158, 2007.
- [42] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *ICDCS '05: Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pages 653–662, Washington, DC, USA, 2005. IEEE Computer Society.
- [43] Liqian Luo, Tarek F. Abdelzaher, Tian He, and John A. Stankovic. EnviroSuite: An environmentally immersive programming framework for sensor networks. *Trans. on Embedded Computing Sys.*, 5(3):543–576, 2006.
- [44] Alan Mainwaring, David Culler, Joseph Polastre, Robert Szewczyk, and John Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM.
- [45] Philo Juang, Hidekazu Oki, Yong Wang, Margaret Martonosi, Li Shiuan Peh, and Daniel Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebrantet. *SIGARCH Comput. Archit. News*, 30(5):96–107, 2002.
- [46] Leo Selavo, Anthony Wood, Qihua Cao, Tamim Sookoor, Hengchang Liu, Aravind Srinivasan, Yafeng Wu, Woonchul Kang, John Stankovic, Don Young, and John Porter. Luster: Wireless sensor network for environmental research. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 103–116, New York, NY, USA, 2007. ACM.
- [47] Anthony Wood, Gilles Virone, Thao Doan, Qihua Cao, Leo Selavo, Yafeng Wu, L. Fang, Zhimin He, Shan Lin, and Jack Stankovic. Alarm-net: Wireless sensor networks for assisted-living and residential monitoring. Technical report, University of Virginia, 2006.
- [48] R.R. Brooks, P. Ramanathan, and A.M. Sayeed. Distributed target classification and tracking in sensor networks. *Proceedings of the IEEE*, 91(8):1163–1171, 2003.
- [49] D. Li, K. Wong, Y. Hu, and A. Sayeed. Detection, classification, and tracking of targets in distributed sensor networks. *IEEE Signal Processing Magazine* '02.
- [50] Deborah Estrin, Ramesh Govindan, John S. Heidemann, and Satish Kumar. Next century challenges: scalable coordination in sensor networks. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 263–270, New York, NY, USA, 1999. ACM.
- [51] A. Zeller. *Why programs fail: a guide to systematic debugging*. Morgan Kaufmann Pub, 2009.
- [52] M.A. Linton. The evolution of dbx. In *Proceedings of the Summer USENIX Conference*, 1990.
- [53] O.S. Center. Xmpi-a run/debug gui for mpi. Technical report, Ohio Supercomputer Center, 1997.
- [54] Shirley Browne, Jack Dongarra, and Anne Trefethen. Numerical libraries and tools for scalable parallel cluster computing. *Int. J. High Perform. Comput. Appl.*, 15(2):175–180, 2001.
- [55] B. P. Miller and J. D. Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*, pages 316–325, Washington, DC, 1988. IEEE Computer Society.

- [56] Jerry Fowler and Willy Zwaenepoel. Causal distributed breakpoints. In *In Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 134–141, 1990.
- [57] Vijay K. Garg and Brian Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5:299–307, 1994.
- [58] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.
- [59] M. Golan and D.R. Hanson. Duel-a very high-level debugging language. In *Proceedings of the Winter USENIX Technical Conference*, pages 107–117, San Diego, January 1993.
- [60] P. Maybee. Ned: The network extensible debugger. In *Proceedings of the Winter USENIX Technical Conference*, 1992.
- [61] P. Winterbottom. Acid: a debugger built from a language. In *Proceedings of the Winter USENIX Technical Conference*, pages 211–222, San Francisco, CA, January 1994.
- [62] Mireille Ducassé. Coca: an automated debugger for c. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 504–513, New York, NY, USA, May 1999. ACM.
- [63] M.L. Powell and M.A. Linton. A database model of debugging. In *SIGSOFT '83: Proceedings of the symposium on High-level debugging*, pages 67–70, New York, NY, USA, 1983. ACM.
- [64] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. Dynamic query-based debugging of object-oriented programs. *Automated Software Engg.*, 10(1):39–74, January 2003.
- [65] Jing Yang, Mary Lou Soffa, Leo Selavo, and Kamin Whitehouse. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 189–203, New York, NY, USA, November 2007. ACM.
- [66] Qing Cao, Tarek Abdelzaher, John Stankovic, Kamin Whitehouse, and Liqian Luo. Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 85–98, New York, NY, USA, November 2008. ACM.
- [67] Arsalan Tavakoli, David Culler, Philip Levis, and Scott Shenke. The case for predicate-oriented debugging of sensornets. In *Proceedings of the 5th Workshop on Hot Topics in Embedded Networked Sensors (HotEmNets)*, Charlottesville, VA, June 2008.
- [68] Mohammad Maifi Hasan Khan, Hieu Khac Le, Hossein Ahmadi, Tarek F. Abdelzaher, and Jiawei Han. Dustminer: troubleshooting interactive complexity bugs in sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 99–112, New York, NY, USA, November 2008. ACM.
- [69] Bor rong Chen, Geoffrey Peterson, Geoffrey Mainland, and Matt Welsh. Livenet: Using passive monitoring to reconstruct sensor network dynamics. In *Proceedings of the 4th IEEE/ACM International Conference on Distributed Computing in Sensor Systems (DCOSS 2008)*, pages 79–98, Santorini Island, Greece, June 2008.
- [70] L. Luo, T. He, G. Zhou, L. Gu, T. F. Abdelzaher, and J. A. Stankovic. Achieving repeatability of asynchronous events in wireless sensor networks with envirolog. In *Infocom*, 2006.

- [71] Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. Sympathy for the sensor network debugger. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 255–267, New York, NY, USA, November 2005. ACM.
- [72] Larry Wittie. The bugnet distributed debugging system. In *EW 2: Proceedings of the 2nd workshop on Making distributed systems work*, pages 1–3, New York, NY, USA, September 1986. ACM.
- [73] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. Comput.*, 36:471–482, 1987.
- [74] Rajib Mall. A novel bi-directional execution approach to debugging distributed programs. In *HiPC '99: Proceedings of the 6th International Conference on High Performance Computing*, pages 95–102, London, UK, December 1999. Springer-Verlag.
- [75] M. Maruyama, M. Maruyama, T. Tsumara, and H. Nakashima. Parallel program debugging based on data-replay. *Transactions of Information Processing Society of Japan*, 46(SIG12(ACS11)):214–224, 2005.
- [76] AD Malony, DH Hammerslag, and DJ Jablonowski. Traceview: a trace visualization tool. In *Proceedings of the First International ACPC Conference on Parallel Computation*, 1991.
- [77] Richard Kilgore and Craig Chase. Re-execution of distributed programs to detect bugs hidden by racing messages. In *In Proceedings of the International Conference on System Sciences*, page 423, 1997.
- [78] Byung-Gon Chun, Kuang Chen, Gunho Lee, Randy H. Katz, and Scott Shenker. D3: declarative distributed debugging. Technical report, UC, Berkeley, 2008.
- [79] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. Framework for instruction-level tracing and analysis of program executions. In *VEE '06: Proceedings of the 2nd international conference on Virtual execution environments*, pages 154–163, New York, NY, USA, June 2006. ACM.
- [80] Bill Lewis. Debugging backwards in time. In *5th Workshop on Automated and Algorithmic Debugging (AADEBUG)*, Ghent, Belgium, September 2003.
- [81] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, Dec. 2002.
- [82] Andreas Zeller. Yesterday, my program worked. today, it does not. why? *SIGSOFT Softw. Eng. Notes*, 24(6):253–267, Nov. 1999.
- [83] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 432–448, New York, NY, USA, October 2004. ACM.
- [84] W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In *ASE '08: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 128–137, Washington, DC, USA, September 2008. IEEE Computer Society.

- [85] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, 1978.
- [86] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.
- [87] C. Fidge. Logical time in distributed computing systems. *Computer*, 24:28–33, 1991.
- [88] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained network time synchronization using reference broadcasts. In *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 147–163, New York, NY, USA, December 2002. ACM.
- [89] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi. The flooding time synchronization protocol. In *Proc. 2nd International Conference on Embedded Networked Sensor Systems*, Baltimore, MD, November 2004.
- [90] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: staged functional programming for sensor networks. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 335–346, New York, NY, USA, September 2008. ACM.
- [91] Ryan Newton and Matt Welsh. Region streams: functional macroprogramming for sensor networks. In *DMSN '04: Proceedings of the 1st international workshop on Data management for sensor networks*, pages 78–87, New York, NY, USA, August 2004. ACM Press.
- [92] Ben Greenstein, Eddie Kohler, and Deborah Estrin. A sensor network application construction kit (snack). In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 69–80, New York, NY, USA, November 2004. ACM.
- [93] A. Awan, S. Jagannathan, and A. Grama. Macroprogramming heterogeneous sensor networks using cosmos. *SIGOPS Oper. Syst. Rev.*, 41(3):159–172, June 2007.
- [94] Ken Birman. A response to cheriton and skeen’s criticism of causal and totally ordered communication. *SIGOPS Oper. Syst. Rev.*, 28(1):11–21, Jan. 1994.
- [95] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with cooja. *Annual IEEE Conference on Local Computer Networks*, 0:641–648, 2006.
- [96] Joakim Eriksson, Adam Dunkels, Niclas Finne, Fredrik sterlind, and Thiemo Voigt. Mspsim – an extensible simulator for msp430-equipped sensor boards. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, JAN 2007.
- [97] Michiel Ronsse, Mark Christiaens, and Koenraad De Bosschere. Cyclic debugging using execution replay. In *ICCS '01: Proceedings of the International Conference on Computational Science-Part II*, pages 851–860, London, UK, May 2001. Springer-Verlag.
- [98] Timothy W. Hnat, Tamim I. Sookoor, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse. Macrolab: a vector-based macroprogramming framework for cyber-physical systems. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 225–238, New York, NY, USA, November 2008. ACM.

- [99] Zonefirst. Zoning design and application guide. <http://www.zonefirst.com/products/DesignManual.pdf>, 2003.
- [100] I.S. Walker. Register Closing Effects on Forced Air Heating System Performance. 2003.
- [101] W. Watts, M. Koplow, A. Redfern, and P. Wright. Application of multizone HVAC control using wireless sensor networks and actuating vent registers. 2007.
- [102] A. Redfern, M. Koplow, and P. Wright. Design architecture for multi-zone HVAC control systems from existing single-zone systems using wireless sensor networks. In *Proceedings of SPIE*, volume 6414, page 64140Y, 2006.
- [103] Tim van Kasteren, Athanasios Noulas, Gwenn Englebienne, and Ben Kröse. Accurate activity recognition in a home setting. In *UbiComp '08: Proceedings of the 10th international conference on Ubiquitous computing*, pages 1–9, New York, NY, USA, 2008. ACM.
- [104] D.B. Crawley, L.K. Lawrie, C.O. Pedersen, F.C. Winkelmann, M.J. Witte, R.K. Strand, R.J. Liesen, W.F. Buhl, YJ Huang, RH Henninger, et al. EnergyPlus: an update. *Proceedings of SimBuild*, pages 4–6, 2004.
- [105] La Crosse Technology Ltd. <http://www.lacrossetechnology.com/>, 2010.
- [106] C. Lin, C. Federspiel, and D. Auslander. Multi-sensor single actuator control of HVAC systems. In *International Conference for Enhanced Building Operations*. Citeseer, 2002.
- [107] I.S. Walker and A.K. Meier. Residential Thermostats: Comfort Controls in California Homes. 2008.
- [108] TRANE. Varitrac changeover bypass vav. <http://www.trane.com/Commercial/Uploads/Pdf/1101/varitrac-6pg.pdf>, April 2003.
- [109] Nielson Kellerman. Kestrel pocket wind meters and weather trackers. www.kestrelweather.com, 2009.
- [110] Bayweb thermostat. <http://www.bayweb.com/>.
- [111] Inc. Energy. T.e.d.: Electricity monitor, energy monitor, power monitor. <http://www.theenergydetective.com/index.html>.
- [112] Airgonomix. Benefits of Micro-Zone HVAC Systems. www.airgonomix.com/AirgonomixWhitePaperZoningBenefits.pdf, August 2008.
- [113] RJ Rose and J. Dozier. EPA program impacts office zoning. *Name: ASHRAE Journal*, 39(1), 1997.
- [114] J.R. Smith, K.P. Fishkin, B. Jiang, A. Mamishev, M. Philipose, A.D. Rea, S. Roy, and K. Sundara-Rajan. Rfid-based techniques for human-activity detection. *Communications of the ACM*, 48(9):39–44, 2005.
- [115] H. Nait-Charif and S.J. McKenna. Activity summarisation and fall detection in a supportive home environment. In *Proceedings of the Pattern Recognition, 17th International Conference on (ICPR'04) Volume*, volume 4, pages 323–326. Citeseer.
- [116] D.H. Wilson and C. Atkeson. Simultaneous tracking and activity recognition (STAR) using many anonymous, binary sensors. In *The Third International Conference on Pervasive Computing*, pages 62–79. Springer, 2005.

- [117] V. Srinivasan, J. Stankovic, and K. Whitehouse. Protecting your daily in-home activity information from a wireless snooping attack. In *Proceedings of the 10th international conference on Ubiquitous computing*, pages 202–211. ACM New York, NY, USA, 2008.
- [118] V. Srinivasan, J. Stankovic, and K. Whitehouse. Using Height Sensors for Biometric Identification in Multi-resident Homes. In *Pervasive*, 2010.
- [119] Thermostat history. <http://www.prothermostats.com/history.php>, 2011.
- [120] U.s. doe residential energy consumption survey. http://www.eia.doe.gov/emeu/recs/recs2005/hc2005_tables/detailed_tables2005.html, April 2008.
- [121] Environmental Protection Agency. Summary of research findings from the programmable thermostat market. Washington, DC: Office of Headquarters, 2004.
- [122] H Sachs. Programmable Thermostats. *ACEEE*, 2004.
- [123] Ge Gao and Kamin Whitehouse. The self-programming thermostat: Optimizing setback schedules based on home occupancy patterns. *First ACM Workshop On Embedded Sensing Systems For Energy-Efficiency In Buildings (BuildSys '09), held in conjunction with ACM SenSys*, November 2009.
- [124] Telkonet smartenergy. <http://www.telkonet.com/index.php>.
- [125] prothermostats.com. Verdant V8-BB-7S Line Voltage Heat Only Thermostat - Programmable.
- [126] Viconics. VT7000 V-PIR Passive Infrared Motion Detector.
- [127] Peco. Smart energy management for classrooms and portables.
- [128] Jiakang Lu, Tamim Sookoor, Ge Gao, Vijay Srinivasan, Brian Holben, John Stankovic, Eric Field, and Kamin Whitehouse. The smart thermostat: Using wireless sensors to increase comfort and save energy. In *SenSys 2010*, 2010.
- [129] M. Gupta, S. Intille, and K. Larson. Adding gps-control to traditional thermostats: An exploration of potential energy savings and design challenges. *Pervasive Computing*, pages 95–114, 2009.
- [130] M.C. Mozer, L. Vidmar, and R.H. Dodier. The neurothermostat: Predictive optimal control of residential heating systems. *Advances in Neural Information Processing Systems*, pages 953–959, 1997.
- [131] C. Federspiel. Wireless mesh networks for energy-conservation retrofits. *HPAC Heating, Piping, AirConditioning Engineering*, 78(11 SUPPL):12–18, 2006.
- [132] Millennial Net. Wireless hvac remote monitoring and control systems. <http://www.millennial.net/industries/hvacmonitoring.php>, 2009.
- [133] Siemens Industry Inc. Apogee building automation system - building automation and control. http://www.buildingtechnologies.siemens.com/BT/US/PRODUCTS__AND__SYSTEMS/BUILDING_AUTOMATION_AND_CONTROL/APOGEE_BUILDING_AUTOMATION_SYSTEM/Pages/apogee_building_automation_system.aspx, 2010.
- [134] Johnson Controls. Personal environments. http://www.johnsoncontrols.com/publish/us/en/products/building_efficiency/integrated_hvac_systems/hvac/personal_environments.html, 2010.

- [135] Kurt W. Roth, Detlef Westphalen, John Dieckmann, Saphir D. Hamilton, and William Goetzler. Energy Consumption Characteristics of Commercial Building HVAC Systems Volume III: Energy Savings Potential. *TIAX LLC*, (4-62-4-71), June 2002.
- [136] N.H. Cohen. Hvac system with energy saving modes set using a security system control panel, April 24 2008. US Patent App. 12/108,644.
- [137] M.L. Simmons and D.J. Gibino. Energy-saving occupancy-controlled heating ventilating and air-conditioning systems for timing and cycling energy within different rooms of buildings having central power units, February 26 2002. US Patent 6,349,883.
- [138] Qucs project. Qucs home page. <http://qucs.sourceforge.net/>, Mar 2011. <http://qucs.sourceforge.net/>.
- [139] Powerhouse Dynamics. Residential emonitor overview. <http://www.powerhousedynamics.com/residential-energy-efficiency/>, 2012.
- [140] P. Haves, L.K. Norford, M. DeSimone, and L. Mei. A standard simulation test bed for the evaluation of control algorithms and strategies. *TRANSACTIONS-AMERICAN SOCIETY OF HEATING REFRIGERATING AND AIR CONDITIONING ENGINEERS*, 104:460–473, 1998.
- [141] O. Ahmed. *Model-based control of laboratory HVAC systems*. University of Wisconsin–Madison, 1996.
- [142] S. Carpenter. Advances in Modeling Thermal Bridges in Building Envelopes. *Enermodal Engineering Limited. Kitchener, ON, Canada*, 2001.
- [143] I. Beausoleil-Morrison. Modelling mixed convection heat transfer at internal building surfaces. In *Building Simulation*, 1999.
- [144] X. Peng and TU Delft. Modeling of indoor thermal conditions for comfort control in buildings. *Delft University of Technology, Delft*, 1996.
- [145] E. Ratnam, T. Campbell, and R. Bradley. Advanced feedback control of indoor air quality using real-time computational fluid dynamics. *ASHRAE Transactions*, 1998.
- [146] B. Birdsall, WF Buhl, KL Ellington, AE Erdem, and FC Winkelmann. Overview of the DOE-2 building energy analysis program, version 2.1 D. Technical report, LBL-19735-Rev. 1, Lawrence Berkeley Lab., CA (USA), 1990.
- [147] H.E. Feustel. COMIS—an international multizone air-flow and contaminant transport model. *Energy and Buildings*, 30(1):3–18, 1999.
- [148] G. Henze, C. Felsmann, and G. Knabe. Evaluation of optimal control for active and passive building thermal storage. *International Journal of Thermal Sciences*, 43(2):173–183, 2004.
- [149] K. Deng, P. Barooah, P. Mehta, and S. Meyn. Building thermal model reduction via aggregation of state. In *American Control Conference*, pages 5118–5123, 2010.
- [150] F. Oldewurtel, A. Parisio, C. Jones, M. Morari, D. Gyalistras, M. Gwerder, V. Stauch, B. Lehmann, and K. Wirth. Energy efficient building climate control using stochastic model predictive control and weather predictions. In *American Control Conference*, pages 5100–5105, 2010.
- [151] Y. Ma, G. Anderson, and F. Borrelli. A distributed predictive control approach to building temperature regulation. In *American Control Conference*, 2011.

- [152] T. Nghiem and G. Pappas. Receding-horizon supervisory control of green buildings. In *American Control Conference*, 2011.
- [153] A. Aswani, N. Master, J. Taneja, D. Culler, and C. Tomlin. Reducing transient and steady state electricity consumption in hvac using learning-based model predictive control. In *Proceedings of the IEEE*, 2011.
- [154] J. Krumm and A. Brush. Learning time-based presence probabilities. *Pervasive Computing*, pages 79–96, 2011.
- [155] A. Alrazgan, A. Nagarajan, A. Brodsky, and N.E. Egge. Learning occupancy prediction models with decision-guidance query language. In *System Sciences (HICSS), 2011 44th Hawaii International Conference on*, pages 1–10, jan. 2011.
- [156] J. Page, D. Robinson, N. Morel, and J.L. Scartezzini. A generalised stochastic model for the simulation of occupant presence. *Energy and Buildings*, 40(2):83–98, 2008.
- [157] R.H. Dodier, G.P. Henze, D.K. Tiller, and X. Guo. Building occupancy detection through sensor belief networks. *Energy and buildings*, 38(9):1033–1043, 2006.
- [158] J. Scott, A.J.B. Brush, J. Krumm, B. Meyers, M. Hazas, S. Hodges, and N. Villar. Preheat: Controlling home heating using occupancy prediction. 2011.