Simplifying Design of Wireless Sensor Networks with Programming Languages, Compilers, and Synthesis

by

Lan Bai

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2011

ACKNOWLEDGEMENTS

I am heartily thankful to my advisor, Robert Dick, for his guidance and support throughout my Ph.D. He sparked my interests in wireless sensor networks and provided valuable advice in the course of this dissertation. He taught me not only by his words but by his personal example how to become an excellent researcher.

I would like to thank Peter Dinda, Lawrence Henschen, and Pai Chou for numerous deep and enlightening discussions. We have closely collaborated on most of the work presented in this dissertation. They have pointed me to insightful references and raised stimulating questions from various perspectives. Pai Chou's group provided the sensor fault measurements used in Chapter VI. I am grateful to other students on our team, David Bild, Scott Miller, Timothy Zwiebel, and Jaime Espinosa, for being the sounding boards for my ideas. I thank Scott Miller for sharing his pool of user study participants.

My thanks must also go to Charles Dowding, Mat Kotowsky, Carl Ebeling, and Michael Hannigan for providing many useful comments from the perspective of application experts. I would like to thank them for sharing their experience and being my first test subjects to evaluate my designs and tools.

I am grateful to the other members on my thesis committee, Prabal Dutta, Jason Flinn, Jerome Lynch, and Zhengya Zhang. They have generously given their time and expertise to better my work.

I would like to thank Oliviu Ghica, developer of the SIDnet-SWANS simulator, for promptly and patiently answering my questions about the simulator and constantly improving his tool. I would also like to thank Yanqi Zhou for working on porting the WASP language to the Eco sensor platform.

My colleagues in our research group have created a collaborative and pleasant working environment. I always find it thought-provoking to discuss research with my labmates: Lei Yang, Zhenyu Gu, Xi Chen, Stephen Tarzia, David Bild, Lide Zhang, Yue Liu, Xuejing He, Yun Xiang, Phil Knag, and Robert Perricone. Many of them are not only great colleagues but also good friends. It is my pleasure to collaborate with Lei on the memory compression work described in Chapter V. I would like to thank Yue and David for sharing their experience and findings about wireless networks. The tools (poles for placing nodes) David built have made my experiments a lot easier.

On a more personal note, I would like to express my gratitude to my friends at Northwestern University and at the University of Michigan, who made my Ph.D. a memorable journey. I would also like to thank Andrew Madden for sharing his LaTeX template, which saved a lot of time formatting my dissertation.

Finally, my special thanks go to my beloved parents. Without their support and encouragement, this dissertation would have not been possible. I would like to thank my mother for helping me with the laborious in-field experiments.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure

ABSTRACT

Wireless sensor networks have opened opportunities for new applications and attracted users from domains beyond computer system design. Sensor network design is challenging. It is generally an ad hoc process carried out by embedded system experts. In this dissertation, we argue that human efforts necessary to the design of sensor networks can be reduced with the help of high-level specification languages, compilers, and synthesis tools. We designed and implemented a framework to simplify and automate the design of a class of sensor network applications. Our results show that a sensor network novice given only a few pages of instructions, can successfully specify sensing applications within 30 minutes, compared with hours or days required by prior approaches. Within approximately 30 minutes, our modeling and design exploration techniques translate these specifications into implementations, automatically selecting from among 405,790 designs. Moreover, our memory management and compiler-assisted techniques make difficult-to-implement optimizations available to novice programmers, enabling better tolerance of sensor faults and making 39% more usable memory available than would otherwise be the case.

We propose a design process that decouples specification from implementation. Application designers specify abstract functionality and design requirements. Compiler and synthesis tools automatically determine implementation details, optimizing design parameter optimization and generating code. First, we develop a design process in which programming novices (e.g., application experts) use high-level, specification languages designed for particular classes of applications. We focus on the class most commonly encountered in sensor network deployment publications. Second, we develop two compiler and run-time techniques to relieve application experts from explicitly dealing with sensor faults and limited memory, two common sources of sensor network design complexity. The first technique automatically generates code for fault detection and error estimation using easy-to-specify hints. The second technique automatically generates code for online memory compression, thereby increasing effective memory. Finally, we develop modeling and optimization techniques to determine high-level design parameters to meet specified design requirements. We present an automated technique that constructs fast and accurate system-level models for sensor networks and an optimization technique that uses these models to rapidly search for the optimal design(s). Our evaluation focuses on homogeneous environments.

CHAPTER I

Introduction

A wireless sensor network consists of spatially distributed autonomous devices, denoted as sensor nodes or motes, that are capable of sensing, computing, communicating with each other wirelessly, and possibly actuating. Sensor nodes are usually small, inexpensive, lightweight, and low power. Although each sensor node is tightly constrained in computation capability, storage capacity, and energy consumption, a large number of these tiny devices can collaboratively execute complex tasks such as object classification and tracking. Wireless sensor networks have opened opportunities for ubiquitous, unobtrusive, and perpetual sensing. As a result, they are natural fits for numerous applications, such as environmental monitoring, infrastructure intelligence, transportation, health care, and surveillance.

Wireless sensor networks empower individuals to gather fine-grained, precise, and extensive information from the physical world. This information can be used to make smart decisions and have timely reactions. Wireless sensor networks will have significant impact on our economy and life with their countless uses. Farmers can enhance quality of their products by planning farming practice according to temperature and soil moisture data gathered from a wireless sensor network deployed in their crops [121]. Scientists can gather valuable data on their study objects leading to new scientific discoveries [108, 142, 136]. Home owners and building facility managers can detect energy waste and plan accordingly to save energy with device-level energy use data gathered by wireless energy meters. A nation under threat of natural disaster can use sensor networks to detect disaster sources and predict its impacts in order to minimize damage. Factories can enable sensing and controlling in locations that previously would have been cost-prohibitive to control industrial process.

With the advances in MEMS sensing technology, low-power computing, and wireless communication, the market of sensor network is expected to grow rapidly in the coming years. Nevertheless, the cost for design and deployment does not decrease as fast as the hardware prices. We anticipate that there will be a greater need for appropriate design tools for wireless sensor networks. They are not only critical for reducing design costs and time-to-market, but also have the potential to open sensor networks to vast users. When wireless sensor networks become widely adopted in various aspects of our lives, more and more people will start to possess and manage wireless sensor networks.

Researchers have devoted tremendous amount of efforts to improve wireless sensor networks by designing low-power hardware components, reliable communication protocols, energy management mechanisms, etc. While existing research has been focused on a bottom-up approach that intends to improve building blocks for wireless sensor networks, we believe that it is also important to take a top-down approach that starts from applications and users and captures high-level design trade-offs. We intend to bridge the gap between existing techniques and potential sensor network users to allow them to efficiently and easily use existing techniques for their applications.

1.1 Challenges of Designing Wireless Sensor Networks

Designing a sensor network is a challenging job. It involves the development of a distributed system composed of resource-constrained and fault-prone devices that interact with each other via unreliable wireless channels. Specifically, a sensor network designer faces the following challenges.

1. A designer needs to convert network-level functionalities and requirements to behaviors of individual sensor nodes. The mapping between node-level performance and network-level performance is usually complex.

2. A sensor network is an "open" system that is largely affected by its deployment environment. The environment not only affects how wireless signals are propagated but also the reliability of sensor nodes. Ignoring environmental effects on component reliability and network reliability leads to performance overestimation.

3. The sensor nodes are usually equipped with limited resources, such as battery energy and memory size. Resource usage should be carefully analyzed and planed. Sometimes, special techniques needs to be used to deal with tight resource constraint, e.g., data compression. However, entangling resource management code with functionality code not only increases complexity of programming, but also increases chances for software bugs.

4. Creating the software that manages applications running on sensor nodes and controls the networks is currently so technically intricate, complex, and laborious that it can take months of work by experienced programmers just to deploy a simple application. Debugging is inherently difficult because it is costly to monitor node states and hard to repeat the same behavior.

5. Designers usually have a handful of system attributes to optimize. When one design parameter is tuned to improve one attribute, it is likely the other attribute will be affected negatively. In other words, they are dealing with a large number of design parameters that are interdependent in a big design space. Design a sensor network requires a proper understanding of the interplay between multiple hardware and software components.

These challenges are so significant that they have slowed deployment plans and tempered initial excitement about wireless sensor network technology. In addition, application experts such as biologists, geologists, and environmental engineers are forced to rely on embedded system experts to implement their ideas. Almost all existing sensor network deployments are implemented by embedded system experts. This approach is costly. Separating design and implementation in this way can also lead to errors due to miscommunication between application experts and embedded system experts. Application experts generally have limited awareness of the constraints on sensor network capabilities imposed by hardware and software limitations. On the other hand, embedded system experts know little about the application requirements, which are tightly related to the measured objects and the working environments. In addition, since application experts' and embedded system experts' domain languages differ significantly, this can cause confusion and misunderstandings that lead to incorrect implementations. Consequently, a collaboration between application experts and embedded system experts requires a large amount of communication, negotiation, redesign, and reimplementation. Wireless sensor networks are considered by potential users because they have the potential to save time and money. When these potential benefits are outweighed by substantial increases in implementation complexity compared to the bulky and expensive, but often easy-to-deploy, sensing solutions already in use, wireless sensor networks will remain unused.

1.2 Towards Wireless Sensor Network Design Automation

We believe that a lot of human efforts in the current design of sensor networks can be eliminated with automated design techniques. Ideally, an intelligent design tool chain that assists any application experts requires no expertise in embedded system design; it lets designers specify what they want instead of how to achieve their goals. An automated design flow takes high-level specifications as inputs and automatically generates detailed, ideally optimal, implementations. The key components of an automated design framework include specification languages in which designers describe their applications and requirements, compiler techniques and synthesis algorithms that transform high-level specifications to low-level implementations, and models that are used to analyze a potential design.

An automated design flow has many advantages. First, it allows efficiently exploring a large design space that contains numerous alternative designs; this is impossible with manual design. In this way, it can generate designs with better qualities than manual designs. Second, it reduces design and development time. Last but not the least, it has the potential to open the design of wireless sensor network to individuals who are not embedded system experts, allowing sensor networks to be quickly adopted in various domains.

Wireless sensor network design is essentially a multi-objective optimization problem. An automated design framework is based on a precise problem formulation. In order to design an appropriate interface between designers and the optimizer, it is important to determine the set of costs or performance metrics application experts care about. For sensor network applications, designers generally care about performance of the network as a whole, instead of individual sensors' behaviors.

The sensor network application space is enormous and constantly expanding. We perceive substantial challenges in designing an unified solution for arbitrary sensor network applications while achieving simplicity in the specification languages. Fortunately, many existing applications have common characteristics. This inspires us to classify the application domain to categories for the purpose to designing separate solutions for each application class. In this dissertation, we attempt to define and solve the design automation problem for a specific class of sensor network applications. Instead of defining an arbitrary class of applications, we favor a systematic approach to categorize the application space. We will start with the most common class of applications, hoping that our approach can be readily used for a substantial class of real-world applications. Our work is a first step towards the automated design of general sensor network applications.

1.3 Dissertation Goal

This dissertation aims to address the key challenges in developing an automated design framework for a class of sensor network applications, including design of specification languages to allow application experts to easily describe their application functionality and requirements, developing compiler and synthesis tools to generate low-level implementation details from high-level specifications, and system-level performance models to efficiently map a potential design to a cost vector. We will formulate the design as an application-oriented problem instead of an implementation-oriented problem. This requires identifying which design aspects falls into the application domain, and which falls into the implementation domain, and more importantly, how to generate the final implementation from the high-level application specification. The first step is to identify a class of applications to focus on based on a systematic categorization of the application domain.

1.4 Dissertation Overview

In Chapter II, we survey existing sensor network applications and categorize the application domain for the purpose of developing compact, special-purpose programming languages for sensor networks. We also present a framework for automated wireless sensor network design.

In Chapter III, we describe a high-level compact language, WASP, and its associated compiler developed for the first archetype. We also present the design and results of user studies to evaluate the designed language and other existing languages. In addition, we describe the specification language for design requirements.

In Chapter IV, we describe our techniques to automatically generate fault detection and error estimation code from high-level specifications.

In Chapter V, we describe compile-time and run-time techniques to increase the amount of usable memory in sensor nodes and other MMU-less embedded systems. Our techniques do not increase hardware cost and require few or no change to existing applications.

In Chapter VI, we describe our approach to automatically generate system-level performance models for sensor networks. We describe how we use this approach to generate system lifetime models considering both battery depletion and node fault processes.

In Chapter VII, we describe a model-based design optimization technique for homogeneous environment. We compare it with a simulation-driven heuristic search. We also discuss challenges and potential solutions for heterogeneous environments.

Finally, we summarize our contributions and present conclusions in Chapter VIII.

Appendix A describes an anomaly in our experiments with the MoteLab testbed.

CHAPTER II

Archetype-Based Design for Sensor Networks

In this chapter, we propose the concepts of sensor network application archetypes and archetype-specific languages. We examine a wide range of wireless sensor networks to develop a taxonomy of seven archetypes. This taxonomy permits the design of compact languages that are appropriate for novice programmers. In addition, we propose a design framework to define the design problem for application experts. Section 2.1 introduces the concept of archetype-based languages. Section 2.2 describes our approach to categorize wireless sensor network applications and presents the archetype taxonomy. Section 2.3 proposes a framework for automating the design process for one application archetype.

2.1 Archetype-Specific Languages

The first step to designing a programming language is to determine the scope of applications it will support. There are two extremes of the range of design philosophies a language designer might adopt: a language might be entirely general-purpose or entirely application-specific. General-purpose languages can be used to specify any application. However, all other things being equal, this flexibility is obtained at the cost of increased language complexity. General-purpose languages have advantages: once such a language is learned, one can write any application with it. However, a novice programmer may never be willing to expend the time to learn it. In contrast, application-specific languages are usually simple and compact, but support only one type of application. This makes it more difficult for a novice programmer to select the appropriate language for an application, and requires the design of numerous languages – one for each type of application. Designers need to learn a new language with each new application. We believe the optimal design philosophy for sensor network programming languages is somewhere between these extremes: a moderate number of specialized languages that together cover most of the sensor network application domain. Ideally, each of these languages should be easy to learn and use for novice programmers.

To find the best tradeoff between the complexity of selecting a language and the complexity of the languages, we propose the concept of *sensor network archetypes*. We have categorized sensor networking applications into archetypes based on functional properties that have large impacts on language design. We have examined a wide range of sensor network applications in order to develop a taxonomy of seven archetypes (see Section 2.2). The language tailored for an archetype is called an *archetype-specific language*.

The taxonomy of sensor network archetypes guides the design of specialized languages for each archetype, these are referred to as *archetype-specific languages*. The concept of archetypes allows templates to be designed to further reduce the programming burden for application experts. In our user study (refer to Section 3.2.2), most test subjects indicated that examples help them to understand a new language. Therefore, we propose the concept of *archetype templates*. These can be generic example programs for specific archetypes or incomplete programs with parameters and lines of code to be modified by programmers according to their needs. An application expert uses an archetype-specific language by reading a short tutorial and using an archetype template to implement an application. We want this procedure to be easy and efficient for novice programmers.

In short, archetype-specific languages have the following advantages.

1. An application expert only needs to learn the language features that are relevant to the application of interest. This reduces required learning and development time.

2. The simplicity of archetype-specific languages permits short tutorials, simple grammars, high levels of abstraction, and productive use of archetype templates. This reduces development time, improves correctness rates, and increases the satisfaction of novice programmers with the design process.

3. The design of high-level languages is simplified by targeting specific groups of applications.

2.2 Taxonomy of Wireless Sensor Network Applications

Specialized, high-level specification languages have the potential to open sensor network design to application experts who are novice programmers. Finding the optimal partitioning of the sensor network application domain for the purpose of language design is challenging. This section describes our study of a wide range of sensor network applications in order to build a taxonomy of sensor network archetypes, and thus languages. Although the sensor network application domain has been studied and categorized before by Röemer and Mattern [116], their results are not directly applicable to our needs. We classify sensor network applications for a different purpose: archetype-based programming language design. We focus solely on application properties that affect the complexity of specification language.

We studied 23 sensor network applications and summarized their application-level requirements and functionalities to extract 19 application properties. These applications, most of which have been deployed, span a wide range of domains: environmental monitoring, structural health monitoring, habitat monitoring, target detection and localization, residential monitoring, active sensing, medical care, farm management, etc. Specifications should focus on the requirements of an application, and avoid implementation details to the greatest degree possible while still maintaining adequate performance. Based on this principle, we identified the following 19 application-level properties (refer to Section 2.2 for definitions): mobility, initiation of sampling process, initiation of data transmission, interactivity, data interpretation, data aggregation, actuation, homogeneity, topography, sampling mode, when sensor locations are known, synchronization, unattended lifetime, mean time to failure, maximum node weight, maximum node size, maximum node volume, maximum node mass, covered area, and quality of service.

Among the 19 application properties, only eight affect the complexity of the specification language. Other properties are constraint-oriented and have little impact on the specification of sensor network functionality. For example, changing the required lifetime of the system from a month to a year will not change the functional specification, although the implementation may change. Specifying constraints can be uniform and straightforward across many application domains, unlike functional specifications. The syntax will be presented in Section 3.3. Therefore, we ruled out these properties as criteria for placing applications. The following eight properties remain.

- **Mobile** indicates whether the sensor nodes are mobile. Mobile nodes may be wearable devices to monitor or track moving objects such as humans and animals [141, 121]. Sensor nodes might also adjust their positions. For applications with mobile sensor nodes, specifications of node localization and node movement control are usually desired. Therefore, mobile sensor network applications require more complex specifications.

- **Initiation of sampling** indicates the condition that causes the nodes to start sampling. It can be periodic, event driven or a mix. Periodic sampling requires specification of the sampling period, while event-driven sampling requires the specification of events.
- **Initiation of data transmission** indicates the condition in which nodes send data through the network. It can be periodic, event driven, or both. Applications for event detection usually require data to be sent to a base station under a certain condition.
- **Actuation** indicates whether the sensor network produces signals to trigger or control other hardware components. For example, the autonomous livestock control application [141] generates stimuli to bulls when the sensor network detects two bulls will soon fight. Actuation requires the specification of triggering conditions and actuation actions, and is therefore more complex than specifying only sensing.
- **Interactivity** indicates whether the network is required to respond to commands sent during operation. Interactions are usually required for initial deployment, reprogramming, maintenance, adjusting operational parameters, and on-site visits. Interactivity requires the specification of commands and reactions.
- **Data interpretation** indicates that in-network data processing is carried out on raw sensor data to filter or compute derivative information. Such online data interpretation may support automated decisions or other actions. Support for data interpretation requires specification of the data processing procedures.
- **Data aggregation** indicates whether data should be aggregated across multiple sensor nodes. Data aggregated requires the ability to specify aggregation algorithms as well as the group of nodes the aggregation operation applies to. For this reason, data aggregation complicates specification.

- **Homogeneity** indicates whether the functionality of every sensor node in the network is the same. For a heterogeneous network, the specification language needs to provide the ability of distinguishing among different types of nodes.

The crossproduct of these eight application attributes results in at least 256 unique points in the language design space. The 23 application samples form 20 points, as shown in Table 2.1. The extreme of designing one language for each point would make it difficult for a user to identify the correct language and increases the burden of language design. Our goal is to find the categorization of sensor network applications that minimizes the complexity of categorizing applications within categories (archetypes) and the complex of using the corresponding language, while also limiting the number of languages required to make the language design process practical.

A good partition should cluster some application types that are adjacent or nearby in the attribute space. In addition, the number of attributes for which multiple dimensions are spanned should be minimized. This suggests using a clustering algorithm for categorization. We adopted the K-Means algorithm to cluster the 23 applications. Dimensions with orthogonal values are treated as sets; dimensions with comparative values are mapped to scalar values with larger values indicate more complex functionality. Choosing the number of clusters involves a trade-off between the complexity of individual languages and the number of languages. The complexity of the specification language corresponding to each application type is hard to quantify precisely ahead of time and the specification language for a potential application category cannot be accurately predicted without language design and evaluation. Therefore, choosing the number of clusters is a somewhat ad-hoc process based on prior experience with sensor network and language design. The resulting clustering-based archetypes are shown in Table 2.2. A row in the table corresponds to one archetype. The "size" column indicates how many applications fit into the corresponding

Table 2.1: Sensor Network Applications

Application	Mobile	Sampling process	Data transmission	Actu-ation	Inter-active	Data interpretation	Data agg.	Homo-geneous
Wisden [102]	N	periodic	periodic	N	N	Y	Y	Y
Habitat [108]	N	periodic	periodic	N	N	N	N	Y
Bridge [57]	N	periodic	periodic	N	N	N	Y	Y
FireWxNet [49]	N	periodic	periodic	N	N	N	N	Y
Light control [123]	N	periodic	periodic	N	N	N	N	Y
ACM [31]	N	periodic	periodic	N	N	N	N	Y
Redwoods [136]	N	periodic	periodic	N	N	N	Y	Y
Surveillance [5]	N	periodic	event	N	Y	Y	Y	Y
VigilNet [45]	N	hybrid	event	N	N	Y	Y	Y
SenSlide [119]	N	periodic	event	N	N	Y	Y	Y
Tracking [118]	N	periodic	event	N	N	Y	Y	Y
Shooter [122]	N	event	event	N	N	Y	Y	Y
Volcanic [142]	N	periodic	event	N	N	Y	N	Y
ElevatorNet [32]	Y	periodic	periodic	N	N	Y	N	Y
ZebraNet [78]	Y	periodic	event	N	N	N	Y	Y
Active sensing [146]	Y	periodic	event	Y	N	Y	Y	Y
Animal control [141]	Y	periodic	periodic	Y	N	Y	N	Y
Farm [121]	Y	periodic	periodic	Y	Y	N	N	N
ALARM-NET [144]	Y	periodic	hybrid	N	Y	N	N	N
CodeBlue [120]	Y	periodic	hybrid	N	Y	Y	N	N
PIPENET [127]	N	hybrid	hybrid	N	Y	Y	Y	Y
NETSHM [22]	N	event	hybrid	Y	Y	N	Y	Y
Tunnel [73]	Y	periodic	event	N	N	Y	Y	N

Table 2.2: Sensor Network Archetypes

Arche-type	Size	Mobility	Sampling	Data transmission	Actu-ation	Inter-active	Data interpretation	Data agg.	Homo-geneous
1	7	stationary	periodic	periodic	N	N	*	*	Y
2	6	stationary	*	event	N	*	Y	*	Y
3	4	mobile	periodic	*	*	N	*	*	Y
4	3	mobile	periodic	*	*	Y	*	N	N
5	1	stationary	hybrid	hybrid	N	Y	Y	Y	Y
6	1	stationary	event	hybrid	Y	Y	N	Y	Y
7	1	mobile	periodic	event	N	N	Y	Y	N

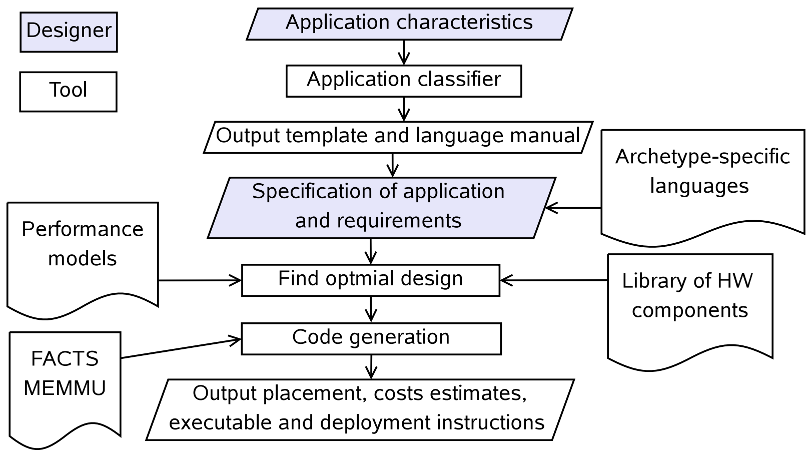

Designer
Tool
Application characteristics
Application classifier
Output template and language manual
Archetype-specific languages
Specification of application and requirements
Performance models
Find optmial design
Library of HW components
Code generation
FACTS MEMMU
Output placement, costs estimates, executable and deployment instructions

Figure 2.1: Automated design flow.

archetype. An archetype is defined by its values in the eight application attributes. "*" means any value is accepted. Note that the specification languages may overlap, i.e., an application may be a member of multiple archetypes.

2.3 A Framework of Automated Design for Sensor Networks

We now propose a framework for fully automated design of wireless sensor networks. It aims to decouple specification from implementation thus minimizing human efforts during the design while allowing exploring a large design space.

Figure 2.1 demonstrates the design flow. Shapes with gray backgrounds indicate designer's responsibilities. Shapes with clear backgrounds indicate the responsibilities of the design tools. An application designer starts with indicating characteristics of his application to the application classifier. These characteristics are listed in Chapter III and are used to determine which archetype an application belongs to. The application classifier selects the archetype according to the designer's inputs and displays the programming template and manual for the corresponding archetype-specific language. The designer then specifies the application-level functionality (a specification language for this purpose is presented in Section 3.2.1) and design requirements (a specification language for this purpose is presented in Section 3.3). The synthesis algorithm then searches the optimal solution in the design space for the given design problem (refer to Chapter VII). During this step, design parameters such as sensor placement, selection of hardware platform, node configuration, battery, etc. are determined. The performance models constructed with techniques described in Chapter VI can be used for quick evaluation of potential solutions. Executables are then generated for the selected platform. During this step, code generation for fault detection, error estimation, and data compression may be used if necessary (refer to Chapter IV and Chapter V). The designer receives the synthesis results: executables, description of placement, along with deployment instructions.

CHAPTER III

High-Level Specification Languages

In this chapter, we present specification languages for application functionality and design requirements. We describe a language (named WASP) and its associated compiler for a commonly encountered archetype identified in Chapter II. We conducted user studies to evaluate the suitability of WASP and several alternatives for novice programmers. To the best of our knowledge, this 56-hour 28-user study is the first to evaluate a broad range of sensor network languages (TinyScript, Tiny-SQL, SwissQM, and TinyTemplate). On average, users of other languages successfully implemented their assigned applications 30.6% of the time. Among the successful completions, the average development time was 21.7 minutes. Users of WASP had an average success rate of 80.6%, and an average development time of 12.1 minutes (an improvement of 44.4%). We also present out definition of the sensor network design problem and describe the specification language for design requirements.

The rest of this chapter is organized as follows. Section 3.2.1 describes the proposed language for the frequently-encountered sensor network archetype. The design of this language is guided by the concept of archetype-specific language proposed in Chapter II. Section 3.1 summarizes prior work on programming languages for sensor networks. Section 3.2.2 and Section 3.2.3 present our evaluation user study and the experimental results.

Section 3.3 presents our definition and language for design requirements. Finally, Section 3.4 concludes this chapter.

3.1 Related Work

Researchers have proposed new sensor network languages to improve design productivity. However, most of these languages have been designed with expert programmers in mind. Although they may improve the productivity of embedded system experts, they are unlikely to make the design and deployment of sensor networks accessible to application experts who are often novice programmers. A few languages have been proposed for application experts. However, their use by novice programmers has not been experimentally evaluated, making it difficult to draw conclusions about their suitability. In this section, we review these languages and summarize the major differences of our work.

Node-level programming languages specify the behavior of each single sensor node. NesC [39] and C are widely used node-level programming languages for sensor networks. Although node-level programming allows manual cross-layer optimizations, they require substantial expertise and effort. These languages are too low-level for novice programmers. In addition, concepts such as events and threads are quite difficult for novice programmers to learn. Efforts [44, 69] have been made to raise the abstraction level of these languages.

Numerous high-level programming languages have been developed for wireless sensor networks to ease their development process. The objective of these languages is to provide appropriate abstractions to hide low-level implementation details from programmers. Network-level programming languages, also called macro-programming languages, let programmers treat the whole network as a single machine [82,95,17,98,9]. Lower-level details such as routing and communication are hidden from programmers. More importantly, they allow programmers to write a distributed sensing application without explicitly managing coordination and state maintenance at the individual node level. Pleiades [59] extends C to achieve a centralized perspective with access to all the nodes in the network via naming. TinyDB and SwissQM allow designers to treat the sensor network as a database and use query languages to extract data from the network [82,95]. Regiment [98] lets programmers view the network as a set of distributed data streams. MacroLab adopts a vector programming abstraction and each vector element corresponds to a node in the network [51]. ATaG [9] is based on data-driven program flow and mixed imperative-declarative specification. It lets developers graphically declare the data flow and connectivity of virtual tasks and specify the functionality of tasks using common imperative language. RuleCaster [17] provides a macroprogramming abstraction with a state-based model and uses a high-level language similar to Prolog.

A few researchers have considered the accessibility of sensor network design to application experts. Some languages [42, 52] are inspired by commercial graphical programming tools such as LabView [62] and Excel. Other researchers made the design of easy-to-use languages tractable by targeting a specific type of application. NETSHM [23] is a sensor network software system for structural health monitoring applications.

We are aware of only two other publication describing experiment evaluation of usability of a sensor network programming language. Eon, which is a programming language proposed for adaptive energy management, has also been evaluated with a user study, but involving only experienced programmers [124]. BASIC was proposed for use in sensor network programming [89]. The authors implemented BASIC for sensor networks and conducted a user study with novice programmers. Their user study is contemporaneous with ours. Their work targeted a different application domain than ours and focused on node-oriented programming.

More comprehensive reviews and comparisons of existing sensor network programming languages can be found in surveys [133, 91]. Mottola and Picco [91] introduced a taxonomy of wireless sensor network programming models. Sugihara and Gupta [133] compared the languages using three metrics: energy-efficiency, scalability, and failure-resilience. They acknowledged that ease of programming is a very important criteria but they believed "criteria of easiness is inherently subjective and the complexity of code largely depends on each application". In contrast, we believe that it is possible and important to evaluate the usability of sensor network languages and have designed and executed a rigorous user study to compare a number of languages.

3.2 WASP: An Example Archetype-Specific Programming Language

We believe that appropriate high-level programming languages and compilers have the potential to make wireless sensor networks accessible to the application experts who have the most to benefit from their use. We propose designing sensor network languages with the novice programmer in mind, hence the following language features are desirable.

1. The languages should support specifying application-level requirements, not just node-level behavior.

2. The languages should not expose low-level implementation details, such as resource management, fault recovery, communication protocols, and optimizations, to users. Users should only need to specify application requirements.

3. The languages should be compact and easy to use. People with limited or no programming experience should be able to almost immediately learn and use them to specify correct sensor network applications.

Once an application's archetype is known, it is possible to provide a program template/example as a starting point. Our studies indicate that the availability of templates improves the success rate for novice programmers implementing sensor network applications from 0% to 8.3% for a node-level language. However, our results suggest that templates are insufficient to make a complex language accessible to novices. Knowledge of an archetype further reduces the burden on a novice programmer because only one archetype-specific language needs to be learned, and each such language is simpler than a general-purpose programming language. We have embodied these language design concepts in a language, called WASP, for a frequently encountered sensor network archetype. In comparison with alternative sensor network programming languages such as TinyScript, TinyDB, and SwissQM, this language results in 1.6× average improvement in success rate and 44.4% average reduction in development time.

This chapter makes the following contributions.

1. We developed a programming language and compiler for the most frequently-encountered archetype.

2. We propose and justify the use of the concept of archetypes to enable the design of compact languages for use by application experts.

3. We conducted user studies to evaluate the proposed programming language and alternative sensor network programming languages. To the best of our knowledge, our 56-hour, 28-user study is the first to evaluate a broad range of sensor network languages.

4. The results of our user study provide insights into the design of programming languages that are accessible to novice programmers.

We selected the archetype with the most existing sensor networking applications as the starting point for archetype-specific language design. This archetype contains the largest number of the applications described in Chapter II. It corresponds to applications that periodically sample and transmit raw data, or filter and aggregate data before transmitting them to a base station from a stationary, homogeneous network. We will refer to this as "Archetype 1". This section presents the proposed language, WASP, as well as its compiler and simulator.

3.2.1 Language Overview

Among the existing languages, those based on database query languages (e.g., SwissQM and TinyDB) provide the most appropriate high-level abstractions for Archetype 1. However, their support for temporal queries may be difficult to grasp for novice programmers, because the database abstraction represents a snapshot of the network that only contains current data (the default table named *sensors*). In order to use historical data, a *storage point* must be explicitly created in the program. The storage point provides a location to store a streaming view of recent data. For example, in TinyDB the code in Figure 3.1 creates a storage point for the most recent eight light samples. For a simple application that compares the current sensor reading with previous readings, developers need to issue a query that joins data from the *sensors* table and the created storage point. Joins require complex query construction that even experienced database users often get wrong. Our experimental results indicate that many novice programmers have great difficulty using joins correctly (see Section 3.2.3 for details). Instead of forcing programmers to explicitly create buffers to store temporal data, WASP makes both historical and current data directly accessible to programmers.

To achieve easy access to both current and historical data, WASP lets programmers

```
CREATE STORAGE POINT
recentlight SIZE 8
AS (SELECT nodeid, light
    FROM sensors
    SAMPLE PERIOD 10s)
```

Figure 3.1: TinyDB storage point.

view the network as distributed data arrays. Each array corresponds to a node-level variable and stores the stream of a particular type of data. Newly sampled data or computed results are inserted at the top of the array, which is indexed from 0. Older data can thus be referenced by indexing into the array. Another major difference between WASP and existing query languages is that WASP lets users specify an application at two levels: node-level and network-level. Operations that only use constants and data generated on one node may be specified at node-level. Data transmission and data aggregation are specified at network-level. The two features permit local data processing while retaining the high-level abstraction that hides the mechanics of routing and communication.

WASP Language Construct

A WASP program is composed of two segments. The node-level code segment, initiated with the keyword “local:”, specifies single node behavior. The network-level code segment, initiated with the keyword “network:”, specifies how data are aggregated through the network and gathered at the base station.

The node-level code segment specifies two types of functionalities: sampling and data processing. The sampling specification indicates the type of sensor data sampled and the associated sampling frequency. The data processing specification indicates how the raw sensed data are processed to generate other data. It may be used for data interpretation, unit conversion, local event detection, etc. The syntax is shown in Figure 3.2. Keywords

```
LOCAL:
SAMPLE sensor EVERY t t_unit INTO buffer
SAMPLE sensor INTO scalar
data_1 = function(args) EVERY t t_unit
data_2 = function(args)
data_3 = arithmetic_expr EVERY t t_unit
data_4 = arithmetic_expr
NETWORK:
COLLECT field1, field2, ...
WHERE node-selection-conditions
GROUP BY node-variable-list
HAVING group-selection-conditions
DELAY t t_unit
```

Figure 3.2: Example WASP template.

are in uppercase. Variables and parameters are in lowercase.

`Sensor` describes the type of sampled data. `Buffer`, `scalar`, and `data_i` are user-defined variables. Programmers can view a variable as an infinite array that stores a time series. Data items in the array can be referred to via indexing. Index 0 represents the most recent datum, while index n represents the nth most recent datum. A data sequence can be referred to using two indices, indicating a range. Fox example, buffer[0:9] returns the most recent 10 elements. Data types of variables are not specified by users, but inferred by the compiler. If a sampling operation or data computation is periodic, `EVERY t t_unit` should be specified at the end of the statement to indicate the period. If absent, this implies that the operation need only be done once. `Function` is selected from a library of built-in aggregation functions used in node-level code. They aggregate data across time on each individual node. The execution order of the statements is determined by the data dependency. Programmers can write them in any order. The syntax of node-level code is designed to be straightforward and readable by novice programmers. The `SAMPLE` clause is similar to English. The other instructions are based on assignment statements that even novice programmers are likely to have used when writing mathematical expressions.

The network-level code segment lets programmers view the entire sensor network as a table and use collective operations to extract desired data. Instead of containing only the current sensor readings, as in TinyDB, this table contains the most recent data for all variables defined in the node-level code segment. Although the table represents a snapshot, its columns may contain variables representing or derived from temporal data. Therefore, only one table exists in WASP; programmers need not create tables or query from multiple tables. Network-level code has a syntax that is similar to the TinySQL language used in TinyDB. It consists of *collect-where-group by-having-delay* clause supporting selection, projection, and aggregation.

WASP has a DELAY statement for specifying maximum data collection latency. The syntax is "`DELAY t t_unit`". Parameter `t t_unit` indicates the maximum delay from data item generation to arrival at the base station. The syntax of the clause for network-level code is more constrained than TinyDB. The data following the *select* keyword can either be a node-level variable or an aggregation function. Expressions are not allowed. In contrast with TinyDB, WASP network-level code does not specify sampling frequency. Frequency should always be specified in the node-level code segment, together with variable definitions. The data transmission frequency can be inferred from the data collection period.

WASP Programming Template and an Example

A template for WASP programs is given in Figure 3.2. Upper-case words are commands. Lower-case words are descriptions of parameters at the corresponding locations; they will be replaced with variables, functions, and expressions by programmers.

We now use an example to demonstrate how to write a sensor network program in

```
LOCAL:
SAMPLE temperature EVERY 10 min INTO tbuf
SAMPLE pressure INTO pbuf
height = pbuf / 100 + 2
temp_level = AVG(tbuf[0:5]) EVERY 1 hour
NETWORK:
SELECT height, AVG(temp_level)
GROUP BY height
```

Figure 3.3: Example WASP code.

WASP. Assume we want to deploy sensor nodes that are able to sense temperature and barometric pressure around a tree to study its microclimate. The nodes sample temperature every 10 minutes. Each node first averages its own temperature samples within one hour, then the average temperatures across nodes are averaged within height levels. The height level of a node is computed from the pressure level as follows: *height = pressure/100 + 2*. Sensor nodes are stationary, so we only need to compute node height levels once. The application is required to sample the average temperature at each height level every hour and transfer the results to the base station.

Compiler and Simulator for WASP

The WASP compiler translates a WASP program into NesC code. The generated NesC code is then compiled to executables with ncc, the NesC compiler for TinyOS. The parser is written with PLY [106], a Python implementation of the compiler construction tools lex and yacc. The implementation of Archetype 1 requires the use of modules for timing, communicating, synchronization, and routing, which we implemented as a library that is automatically accessed by the generated code. We constructed a NesC template for Archetype 1 that embodies the partial implementation required for any application in the archetype. The Collection Tree Protocol (CTP), implemented as TinyOS components, is used for the routing and data collection. In the template, application-dependent code segments are marked with special symbols, which are replaced with NesC statements generated by the WASP compiler. The replacement is automated with a Python script. During compilation, variables in the WASP program are converted to arrays or scalars with explicit data types and the minimum sizes of arrays are computed. The sampling instructions are converted to NesC instructions to control the sensor components. Other node-level instructions are converted to tasks. The period specification in the WASP program is converted to instructions to set timers. The network-level code is converted to data transmission and in-network data aggregation instructions in NesC. The compiler has been tested with the three applications from our user study (refer to Section 3.2.2 for more details). The generated code was run on a multi-hop network composed of four TelosB nodes. We did not yet work on compiler optimization of performance and power.

To support our user study, we also implemented a discrete event simulator for WASP in Python. The parser is modified to generate Python code that creates sampling, processing, and data collection events. The simulator is only used to check functional specification, not implementation or reliability. Therefore, it emulates a perfect network: every operation is instantaneous; there is no node failure or communication loss. The sensor readings are randomly generated in the range from 0 to 1,023. A user interface was also developed for WASP, providing a simple programming environment for editing, saving, compiling, and simulating WASP programs.

3.2.2 User Study

To determine the impact of using WASP on programmer success rates and development times, and to assess the value of archetype-specific languages, we conducted a user study that tested 28 novice programmers using five different programming languages. This section describes the protocol of our user study. The materials used in the study are available on our project website [4].

Questions

The user study was designed to address these questions:

1. What impact does the use of specialized languages have on programmer productivity, as quantified by success rate and development time?
2. What impact does the use of programming templates have on productivity?
3. Is the node-level or the network-level programming model better for novice programmers?
4. Can novice programmers efficiently and correctly use WASP?
5. What is the most appropriate language for the most frequently encountered archetype?
6. What are the primary difficulties novice wireless sensor network programmers have with programming languages?

Languages Under Test

We used the following criteria when selecting languages for testing and comparison: (1) the language is designed to simplify sensor network programming and it provides high-level abstractions; (2) it was designed to support applications that carry out periodic data sampling and transmission, i.e., Archetype 1; (3) it has been implemented and the associate tool chain is publicly available. Five programming languages were selected for comparison: TinyScript, TinyTemplate, TinySQL, SwissQM, and WASP. Three of them (TinyScript, TinySQL, and SwissQM) are from existing work with released software tools.

TinyScript [69] is a general-purpose, node-level, event-driven programming language used for the Maté virtual machine [68]. Programmers write imperative code for event handlers. We made two major changes to create a specialized version of TinyScript, called TinyTemplate, for the most frequently encountered archetype. First, we pruned the library and handlers of TinyScript to only contain functions and events that are related to the target archetype. Second, we provided a programming template. The template is a parameterized example program that implements periodic sampling and data aggregation; comments in the program indicate the variables and instructions that should be replaced for different applications. Expecting that it will be extremely difficult for novice programmers to implement multi-hop communication within reasonable a amount of time, we let the test subjects of TinyTemplate and TinyScript assume a one-hop network structure, in which every node can directly communicate with the root node. Even so, the success rates were extremely low for these two languages.

TinySQL [82] is the SQL-like language used in TinyDB. Programmers view the whole network as a table, with each row indexed by node identification number. User-defined storage points are used in this language to buffer temporal data. SwissQM [95][1] is a programming interface for a query virtual machine. The query language for SwissQM is similar to TinySQL, but instead of letting users write textual code, SwissQM provides a graphical interface. The interface makes composing queries convenient, but it also constrains the supported applications; temporal queries cannot be supported by this interface.

In our study, it was our goal to compare languages and minimize the effect of other factors such as documentation and programming environment. We therefore rewrote the tutorials for these languages (these tutorials are available at our project website [4]). The published documents [94, 81, 70] were generally written for programming experts and proved to be very difficult for novice programmers to understand. Programming templates

[1]The new version had not been released at the time we designed our experiments; version 1.0 was used. This is unlikely to have significantly effected results. Although the new version of SwissQM allows users to write query code, temporal queries are not supported.

were provided for WASP, TinySQL, and TinyTemplate. The graphical interface of SwissQM is considered to be a template.

In practice, system design and programming are interactive and iterative processes. Hence, feedback to test subjects is necessary for the user study to approximate real-world circumstances. Asking users to work with a collection of sensor nodes has the potential to introduce problems that are orthogonal to language design and thereby reduce the discerning power of the study. In order to focus on measuring the impact of the language on productively writing functionally correct code, we associated each language with a simulator. A network composed of four nodes was simulated for each language. These simulators run in real-time, and emulate ideal sensor networks without delay or failure. The TinyOS simulator, TOSSIM [71], was used for TinyScript, TinyTemplate, and SwissQM. We implemented a simulator in Python for TinySQL[2]. The TinySQL code is translated into iterative database queries that are passed to a database server. The creation of storage points is converted to creation of view points. We implemented a discrete event simulator for WASP in Python. Though the implementation of the simulation environments for these languages differ, the user interfaces are quite similar.

User Recruitment

Our 28 test subjects are from a variety of fields: science, engineering, arts, etc. Ten of them have no programming experience. The others, mostly students in engineering fields, have different levels of experience with Fortan, C, and Matlab. We claim that the level of programming experience for this population is representative of sensor network application experts.

[2]We could not get TinyDB working and the released tool does not support the semantics for temporal query, so we wrote a new simulator for it.

Study Structure

Our study procedure is designed to permit fair comparisons among languages while maintaining short duration studies. Each of the five languages, except SwissQM, is evaluated based on use by five novice programmers. SwissQM cannot support Task 3 so it was only tested with three test subjects. By randomly assigning languages to participants, each language was tested by a combination of participants with different background and programming experience. First, the test subjects are introduced to wireless sensor networks via a short description. This gives test subjects a basis for understanding the programming languages and tasks. Next, the test subjects are given 30 minutes to read a tutorial for the language under test, and to familiarize themselves with the programming environment.

After that, they are given the description of two sensor network programming tasks; 40 minutes are permitted for each one. The description of the second task is given after the first is complete, or after 40 minutes have elapsed. The test subjects were permitted to notify the test administrator when they think they have a correct solution. Finally, test subjects answer a survey to provide feedback on the language, tasks, programming environment, etc. The screen is recorded during the study, allowing us to examine the interaction between the programmers and the programming environments. During the study, the test subjects are permitted to ask the test administrator questions about the tutorials or the task descriptions. However, the administrator does not answer questions related to the implementation of the tasks. Though we used three tasks and five languages for the user study, each test subject was asked to complete only two tasks in one language, in order to keep the study short enough for participants. The selection of language and tasks for each test subject was random. To eliminate ordering effects, we randomized the order of the tasks.

Tasks

We selected three tasks that are representative of the target archetype and span different levels of difficulty. These tasks are closely related to the real deployed sensor network applications. Task 1 is a basic environmental monitoring application that transmits raw sensor data to a base station. Task 2 requires node grouping and data aggregation. Task 3 requires temporal processing. SwissQM is inherently unable to support Task 3. The descriptions of the tasks, which are identical to those provided to study participants, follow.

- Task 1: Sample light and temperature every 2 seconds from all the nodes in the network. Transmit the samples with their node identification numbers to the base station [108, 49, 31].
- Task 2: Sample light and temperature every 3 seconds from all the nodes in the network. Collect average temperature readings from nodes that have the same light level. Light levels are computed by dividing raw light readings by 100 [136].
- Task 3: Sample temperature every 2 seconds from all the nodes in the network. Transmit the node identification numbers and the most recent temperature readings from nodes where the current temperature exceeds 1.1 times the maximum temperature reading during the preceding 10 seconds.

3.2.3 Experimental Results

The user study evaluated five programming languages when used by 28 novice programmers. This section presents and analyzes the study results.

Results of User Study

We used *success rate* and *time-to-success* to quantify programming productivity. Table 3.1 shows the success rate and the average time-to-success for each language and task.

Table 3.1: Results of User Study

Language	Success rate			Develop time (min)			User feedback (0–7)				
	T1	T2	T3	T1	T2	T3	Tutorial	Task	Env.	Understand	Easy
SwissQM	3/3	3/3	N.A.	5.7	11.3	N.A.	5.7	5.7	5	5.7	6
WASP	2/2	2/4	2/4	16	31	29.5	4.4	5.4	5.8	4.2	4.6
TinySQL	3/4	2/3	0/3	17.7	27.5	N.A.	4.6	5.8	5.6	4	4.8
TinyTemplate	1/4	0/3	0/3	34	N.A.	N.A.	5.2	5.6	4.6	2.8	3.2
TinyScript	0/3	0/3	0/4	N.A.	N.A.	N.A.	4.4	5.4	4.2	3.8	3.2
WASP2*	3/3	3/4	2/3	3	9.7	23.5	4.4	6.2	5.8	5.2	4.8

*WASP2 is presented in Figure 3.2.3.

The success rate is shown in the form of n/m, meaning n participants out of m succeeded within 40 minutes. The third column shows the average time-to-success. The success rates of TinyScript and TinyTemplate were low; only one test subject completed the simplest task with TinyTemplate. TinySQL and WASP have similar productivity for Tasks 1 and 2, but differ for Task 3. None of the test subjects completed Task 3 with TinySQL, while two out of four succeeded at Task 3 with WASP. The average success rate for TinySQL is 47.2%. The average success rate of WASP is 66.7%. SwissQM has 100% success rate and the shortest completion time for Tasks 1 and 2. However, it does not support Task 3.

The failure of test subjects to complete any tasks with TinyScript suggests that TinyScript may be less appropriate for novice programmers than the other types of languages evaluated. When reviewing mistakes programmers made during the study, we observed two significant obstacles encountered by TinyTemplate users. First, users had trouble with the event-driven programming model; most participants ended up using the wrong event handlers. In addition, many users fell back to programming in an imperative manner. They attempted to specify periodic events with a for loop. Second, novice programmers had trouble implementing node communication. None of the TinyScript users wrote the piece of code required for communication between two nodes (this requires calling the *bcast* function to send the data in a buffer, and calling the *broadcast* function in the broadcast handler to retrieve the data). These issues hindered most programmers from further understanding the programming template provided in TinyTemplate. The difficulty of using TinyScript for novice programmers was also demonstrated in the BASIC study [89] by Miller et al., which used simpler tasks. The increased success rate with TinyTemplate relative to TinyScript suggests that a programming template can ease development with a general-purpose language. It is intuitive for WASP and TinySQL to have similar results for Task 1 and Task 2, as the solutions for these tasks are similar in both languages. WASP improved the success rate of Task 3 from 0 to 2/4 because it allows simpler semantics for accessing temporal data. The short development times of SwissQM for Tasks 1 and 2 can be attributed to two features: (1) SwissQM is well specialized to these tasks and (2) SwissQM provides an easy-to-use graphical user interface.

In addition to the objective metrics of success rate and time-to-success, we also asked participants to provide their feedback on the study-related factors. They were asked to use a number, on the scale from 1 to 7, to indicate their agreement with the following statements (1 means strongly disagree and 7 means strongly agree):

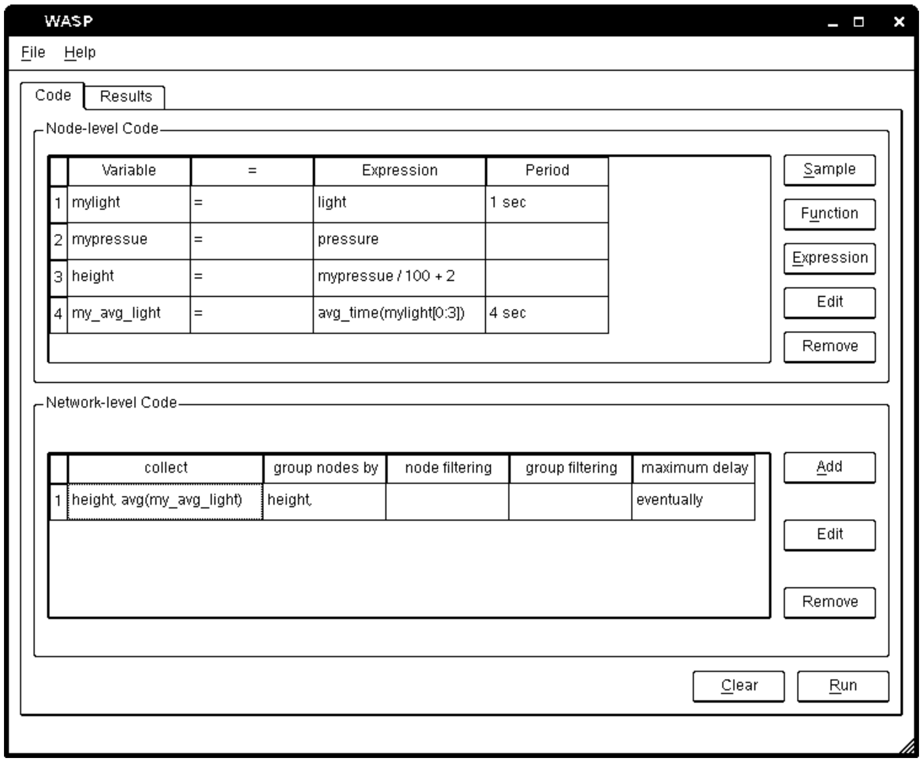
1. The tutorial is easy to understand.
2. The descriptions of the tasks are clear.
3. The programming environment is friendly.
4. I understand this programming language.
5. The programming language is easy to learn and use.

Statements 1, 2, and 3 help us to identify problems caused by difficulty in understanding the manual or the task descriptions. They also provide additional evidence that may help to better understand the success rate and time-to-success results. For example, if users of one language had difficulty understanding the given tasks but users of another language understood the tasks well, we could not conclude that the difference in productivity is due to the accessibility of these languages. We tried our best to avoid bias caused by factors other than the languages, although we could not totally eliminate biases resulting from differences in the released tool chains of existing languages.

The user feedback columns in Table 3.1 show the average ratings for each language for the aforementioned five questions. The five rightmost columns correspond to the statements listed above. According to the results, all participants understood the tutorials and tasks. TinyScript and TinyTemplate have less friendly programming environments, but this is not the major reason for their low success rates. Screen recordings did not reveal that participants encountered any particular difficulty with the programming interface. Statements (4) and (5) focus on user experience with the languages. Contrary to our expectations, the TinyScript users think they understand the language better than TinyTemplate users. This may seem counter-intuitive because the TinyTemplate language is a simplified version of TinyScript. The simplest explanation we have found for this irregularity is that TinyScript users are over confident in their understanding of the language. There is some evidence for this conclusion; several TinyScript users made the error of using the *uart* function to send data to the base station. On the contrary, the TinyTemplate users were able to see that the correct implementation for data collection is more complicated, thanks to the template. The ratings for the last statement order languages by perceived difficulty. This ordering is consistent with actual productivity: in general, the harder a language is perceived to be, the lower its success rate.

Enhanced Version of WASP

From the user study, we observed that, although appropriate programming idioms for Tasks 1 and 2 are similar for SwissQM and WASP, the average completion time for WASP



	Variable	=	Expression	Period
1	mylight	=	light	1 sec
2	mypressue	=	pressure	
3	height	=	mypressue / 100 + 2	
4	my_avg_light	=	avg_time(mylight[0:3])	4 sec

	collect	group nodes by	node filtering	group filtering	maximum delay
1	height, avg(my_avg_light)	height,			eventually

Figure 3.4: User interface of WASP2.

is almost 3× that of SwissQM. By studying screen recordings, we found that test subjects spent a significant amount of time locating and correcting syntax errors in WASP. This was not the case for SwissQM because its interface prohibits many syntax errors; part of the programming is done via selecting from lists and checking radio buttons. The WASP interface provides a text window into which arbitrary text may be entered to compose a program. Program errors are not detected until the syntax check or simulation is started by the users by clicking the associated button. To investigate the impact of user interface on sensor network programming by novice programmers, we designed WASP2. WASP2 is linguistically similar to WASP, but the user interface has a number of enhancements.

Instead of letting users input arbitrary code, WASP2 provides dialogs for composing different types of instructions. The dialogs are equipped with lists containing already-defined variables, validators, auto-completers, syntax checkers, and pop-up warning mes-

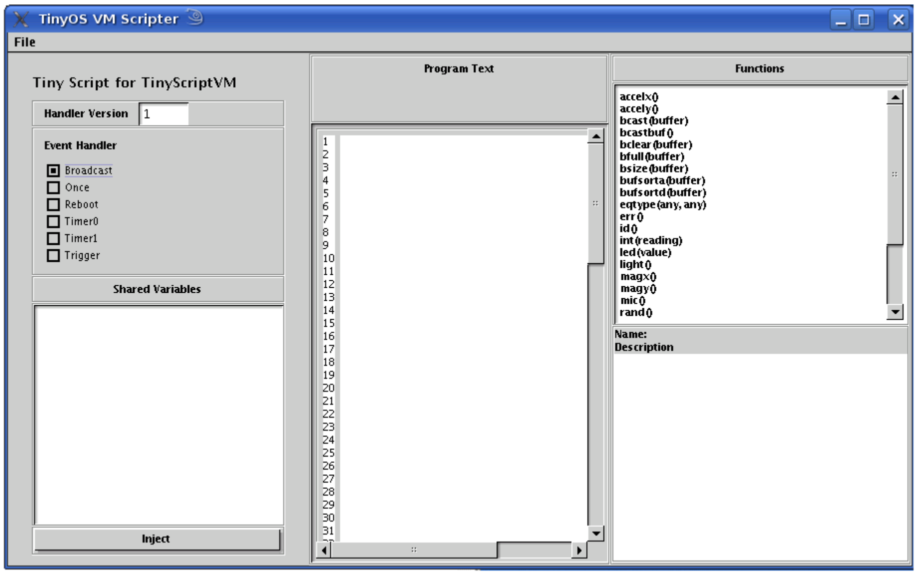


Figure 3.5: User interface of TinyScript.

sages to accelerate instruction editing, prevent user error, and detect syntax errors as early as possible. The widgets in the interface are tagged with pop-up windows displaying appropriate descriptions and help so that programmers do not need to frequently refer to the tutorials. The WASP2 interface is shown in Figure 3.4. We repeated the original study protocol for WASP2 with another five novice programmers. The results are shown in Table 3.1. Compared with WASP, WASP2 improves success rate by 20.8%, and reduces average development time by 58.2%. WASP2 results in better productivity than SwissQM: 20.8% improvement in success rate and 25.3% reduction in average development time for Tasks 1 and 2 (recall that SwissQM does not permit the temporal queries required for some tasks in Archetype 1, e.g., Task 3).

Useful Language Features

Our observations during the user study, and its results, suggest that the following aspects of our sensor network programming language are most useful in improving programmer productivity.

- The network-level abstraction reduces programming effort; it allows programmers to apply operations on data located on different sensor nodes without being concerned about where the operations are executed and how the data are migrated in the network. This feature also distinguishes SwissQM, TinySQL, and WASP from TinyTempalte and TinyScript. The latter two have low success rates partially because the programmers need to explicitly implement node-level communication.

- The combination of network-level and node-level programming was helpful instead of confusing. All test subjects of WASP and WASP2 easily identified which code segments they should use for different data processing operations. Node-level programming allows programmers to specify temporal data processing more easily compared to TinySQL.

- Programming templates helped programmers understand and use the language more easily. Many test subjects indicated that the templates were helpful and created their programs by modifying the template. Note that templates are only useful when the underlying language provides appropriate abstractions for sensor networks. For example, although we provided a template for TinyScript (TinyTemplate), most test subjects still have trouble understanding the language.

- The enhanced graphical user interface that enables guided and constrained programming greatly improves productivity; it reduces the chance for programmers to make grammar errors and assists them in identifying programming errors. Note that this feature was effective only when the key concept of the language could be grasped by the programmers. Even though TinyScript also provides a graphical user interface with similar functionality (Figure 3.5), it is still difficult to use due to the barrier introduced by the node-level and event-driven programming style.

In summary, an appropriate abstraction model of a sensor network is the key factor for a programming language to be accessible to novice programmers. Based on such an abstraction, programming template and well-designed user interface can further improve programming efficiency.

Statistical Analysis of the Results

The goal of the user study is to compare programming languages in terms of success rate and completion time. Examining Table 3.1 is of some value, but it is possible that some of the observed trends could be the result of random variation. In this section, we use statistical tests to determine which trends are statistically significant.

Some languages may have higher success rates and shorter completion times than other languages. Statistical tests are used to determine the probability of these hypotheses being correct, based on a limited set of measurements. The success rates and completion times for one language have distributions. We will test the probability that the means of the distributions for one language exceed those of another. The hypothesis we hope to support is called *alternative hypothesis*. An example alternative hypothesis is "the mean completion time using WASP2 is shorter than that using WASP". The corresponding contradictory *null hypothesis* is "the mean completion time of WASP and WASP2 are the same." The outcome of a statistical test is to reject or not reject the null hypothesis. We will apply both the two-tailed unmatched paired t-test and rank sum test [88].

The t-test and rank sum test are used to compare two populations of independent random samples. The t-test assumes the populations have Gaussian distributions, while rank sum test does not make assumption about the distributions. Generally, with stronger assumptions, we can draw stronger conclusions. Weaker assumptions allow higher confidence in the conclusions, but allow fewer conclusions. We suspect, but are uncertain, that

Table 3.2: Statistical Test of Success Rate

Languages	T-test	Rank sum
WASP vs. TinySQL	0.542	1.000
WASP2 vs. WASP	0.511	0.600
WASP vs. TinyTemplate	**0.035**	0.100
WASP vs. TinyScript	**0.016**	0.100

the distributions are normally distributed. Therefore, we show the implications of both statistical tests. The t-test and rank sum test both assume that samples are independent of each other. In our case, one sample corresponds to the results of one user implementing one task. In our user study, each test subject completed two tasks. We assume that the ordering effects (doing the first task may potentially improve the user's performance on the second one) of tasks assigned to the same test subject is negligible. Therefore, the two tasks for the same individual are treated as two independent samples. The observed significance value (p-value) is used to report the extent to which the test statistic agrees with the null hypothesis. A smaller p implies more evidence to reject the null hypothesis.

T-Test on Success Rate

The success rate is computed as the percentage of test subjects who have completed a particular task using a particular language. In this case, each language has three samples corresponding to the three tasks. The p values are shown in Table 3.2. "A vs. B" in the first column corresponds to the hypothesis "the mean of success rate for language A is larger than that of B".

Rank Sum Test on Success Rate

This test assumes only that samples are independent. The results are shown in Table 3.2. The results indicate that, with high confidence, we can conclude (1) WASP has a higher success rate than TinyScript and TinyTemplate; (2) the mean completion time for Task 1 using SwissQM is longer than that using WASP2; and (3) the mean completion time for Task 2 using WASP is longer than that using SwissQM.

T-Test on Completion Time

Completion times for the three tasks are separately considered because the tasks have different complexities. Mixing samples of different tasks would introduce bias because it was not guaranteed that each language was tested with the same set of tasks. This test is for the hypothesis that, on average, programmers require more time to complete the task with one language, than with another. There are two caveats. First, the t-test assumes that samples are from normal distributions and the standard deviations of the two distributions are the same. Second, due to the time limit of the study, completion times for users who had not finished within 40 minutes are unknown. Therefore, we estimate the completion times for all failed cases to be 50 minutes. This number is a lower bound obtained via investigating the screen record of all failed tests. Our estimation is conservative and it may make difficult languages look easier than they are[3]. The p values are shown in Table 3.3. "A vs. B" in the second column corresponds to the hypothesis "the mean of completion time for language A is larger than that of language B". The statistically significant results are highlighted.

Rank Sum Test on Completion Time

The rank sum test does not make any assumption about the distribution of the completion times. Ranks are assigned to samples in order of decreasing completion times. Samples corresponding to incorrect implementations are given equal ranks that are higher (worse) than those of all correct implementations. The p values are shown in Table 3.3.

[3]This may introduce bias, but will bias against the hypothesis that our programming language is easier to use than others.

Table 3.3: Statistical Test of Completion Time

Task	Languages	T-test	Rank sum
1	TinySQL vs. WASP	0.616	0.667
	TinyScript vs. WASP	**0.026**	0.200
	TinyTemp vs. WASP	**0.029**	0.133
	WASP vs. WASP2	0.212	0.200
	TinyScript vs. TinyTemp	0.437	1.000
	WASP vs. SwissQM	0.299	0.800
	SwissQM vs. WASP2	**0.039**	0.100
2	TinySQL vs. WASP	0.600	0.914
	TinyScript vs. WASP	0.212	0.571
	TinyTemp vs. WASP	0.213	0.571
	WASP vs. WASP2	0.123	0.200
	WASP vs. SwissQM	0.083	0.067
	SwissQM vs. WASP2	0.517	0.914
3	TinySQL vs. WASP	0.289	0.571
	TinyScript vs. WASP	0.210	0.429
	TinyTemp vs. WASP	0.289	0.571
	WASP vs. WASP2	0.572	0.571

3.2.4 Lessons Learned

Conducting user studies allowed us to test our hypothesis, evaluate and improve our design, and develop new ideas that would not occurred to us if we were isolated from users. Unfortunately, user studies quickly consume budgets and time, so it is desirable to carefully design them. This section summarizes the lessons we learned and our observations.

- Conduct a small-scale test study first to minimize unexpected problems in the later, large-scale study. A test study needs not be as strict as the formal study. We tested several tasks with EECS students and our collaborators before the study, though the final study requires randomly recruited strangers who are not programmers. Our test study revealed some bugs in the programming tools and helped us determine a reasonable time limit for the large-scale study.

- Observe the user and record the study without interrupting or intervening. Having a record of the user's behavior allows later analysis of anomalies. During our study, the screen was recorded, the observer took notes on the questions users asked, and users could sketch on draft papers and or mark anything in the language manual. These records allowed us to determine where users become stuck and what mistakes they make.

- Try to consider all factors that may affect your study results. We were interested in the differences among programming languages, so we eliminated the effects introduced by the working environment by using the same editor and similar simulators for every test.
- End user behavior can be very different from designers expectations.
- Programming examples are helpful to novice programmers. Many test subjects indicated that they found the templates helpful and most of them suggested including more examples in the language tutorials.
- Event-driven models, explicit programming data communication, and table joins are hard for novice programmers to grasp.

3.3 Specification Languages for Design Costs and Requirements

The WASP language only lets designers specify the functionality of a sensor network application. An application specified with WASP can map to multiple implementations that have different performance and costs. A designer usually have a handful of attributes to optimize; the attributes are used to determine whether one design is more preferable than another. Different applications may have different requirements. For example, a sensor network deployed to monitor a wild fire is expected to function on battery for at least three months during the fire season, while a sensor network deployed to monitors a

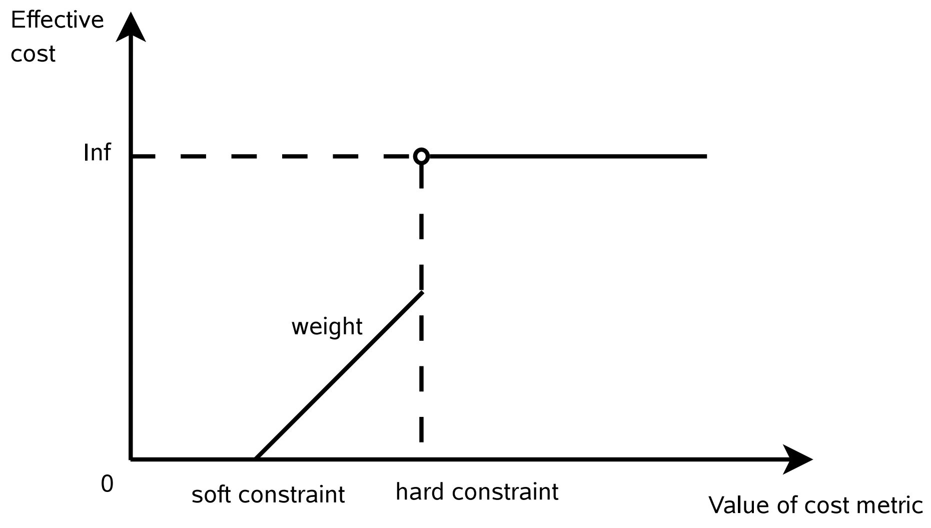


Figure 3.6: Relation between effective cost and value of a cost metric.

bridge is expected to have a lifetime as long as possible. The specification language of application requirements lets the designer describe their criteria for an optimal design.

The design requirement specification is based on the definition of the design problem. The problem definition, as well as the specification language, should be general flexible to support various application requirements, simple enough to allow application expert to understand and use, and precise enough for the design tool to determine an optimal design. In addition, the set of design attributes should represent the interests of the application experts, instead of requiring the designer to manually convert ones design goal to another criteria. For example, an application experts should be able to directly specify the required unattended lifetime of a sensor network, instead of requiring them to convert that into the power consumption of a sensor node.

The specification for application requirements is composed of a list of specifications for different cost metrics, in the form of "COST soft-constraint hard-constraint weight". COST is the name of a cost metric, selected from a pre-defined set of cost metrics. *Soft constraint* is the optimization threshold. It means that when COST is lower than *soft* *constraint*, there is no need to optimize it. *Hard constraint* specifies a threshold the COST must not exceed. A solution with a COST larger than *hard constraint* is considered as infeasible. *Weight* suggests how significant the COST contributes to the total cost. In other words, if COST has to be worse than *soft constraint* then optimize it with the specified weight. Weight (in the range of 0 1) indicates the importance of different optimization objectives when there are multiple. Figure 3.6 demonstrates the meaning of a specification with a figure plotting the effective cost as a function of a single cost metric. When the cost metric is smaller than *soft constraint*, its contribute to the effective cost is zero. When it is larger than *soft constraint*, the effective cost linearly increases with a rate equal to *weight*. When it exceeds *hard constraint*, it is infinite.

When *soft constraint* is 0 and *hard constraint* is Inf (infinite), the corresponding cost metric becomes a pure optimization objective. When *hard constraint* equals *soft constraint* or *weight* is zero, it becomes a pure constraint. For example, "PRICE 100 1000 0.8" means that the total price must be lower than 1000, meanwhile, if it has to be larger than 100, then try to minimize it with weight 0.8. If the designers does not include a cost metric in the specification, it means that it needs not to be optimized. It is essentially the same as "COST Inf 0 0". The specification for value metrics that are to be maximized is the same.

We leave the discussion of how the design tool constructs a single optimization objective as a function of individual costs to Chapter VII.

3.4 Conclusions

Application-level specifications languages that are accessible to novice programmers have the potential to open sensor network design to application experts. In this chapter, we have described a high-level language for the most frequently encountered archetype. Our user study of 28 novice programmers using five programming languages indicates that archetype-specific languages have the potential to substantially improve the success rates and reduce programming times for novice programmers compared with existing general-purpose and/or node-level sensor network programming languages. Our language, WASP, increased the success rate by 1.6$\times$ and reduced average development time by 44.4% compared to other languages. We have also described a definition of a sensor network design problem and the corresponding specification language for design requirements.

CHAPTER IV

Simplified Programming of Faulty Sensor Networks via Code Transformation and Run-Time Interval Computation

Detecting and reacting to faults is an indispensable capability for many wireless sensor network applications. Unfortunately, implementing fault detection and error correction algorithms is challenging. Programming languages and fault tolerance mechanisms for sensor networks have historically been designed in isolation. In this chapter, we describe an extension to the WASP language described in Chapter III to simplify the design of fault-tolerant sensor networks. We describe a system to make it unnecessary for sensor network application developers and users to understand the intricate implementation details of fault detection and tolerance techniques, while still using their domain knowledge to generate effective fault detection, error correction, and error estimation mechanisms. The FACTS system we have developed translates low-level faults into their consequences for application-level data quality, i.e., consequences application experts can appreciate and understand. We implement this system by extending our sensor network programming language, WASP; its compiler and runtime libraries are modified to support automatic generation of code for on-line fault detection and tolerance. This code determines the impacts of faults on the accuracies of the results of potentially complex data aggregation and analysis expressions. We evaluate the overhead of the proposed system on code size, memory use, and the accuracy improvements for data analysis expressions using a small experimental testbed and simulations of large-scale networks.

4.1 Introduction

It is important for a sensor network program to be capable of detecting and reacting to faults. Sensor nodes are composed of fault-prone components and they often operate in harsh and time-varying environments. Experience from prior deployments [134,63,49,13] has demonstrated that deployed nodes can fail or produce erroneous results. A fault is an incorrect state of hardware or software resulting from failures of components, physical interference from the environment, or incorrect design. A sensor node may experience a fault when water leaks through its package and damages sensors. A network communication link may experience a fault in the presence of radio interference. A system failure occurs when faults prevent the system from providing a required service. Embedding fault detection, fault recovery, and error estimation functionalities in a program make it more robust and allow more accurate interpretation of data gathered by the application.

There are many challenges to detecting and reacting to faults in a wireless sensor network. (1) Distributed system architectures generally increase the difficulty of fault tolerance. (2) Fault detection and correction requires communication, which generally imposes high energy overhead in wireless sensor networks. (3) Fault detection and correction algorithms increase program complexity, thus leading to a higher probability of software errors. (4) Different applications may have different fault tolerance requirements and manifestations.

Many researchers have proposed methods for fault detection, fault correction, and network diagnosis for sensor networks [60,26,76,77,112,99,47]. These technologies may be readily used by experienced sensor network developers. However, it requires tremendous effort for novice programmers to learn and use these techniques, especially in the context of wireless sensor network design.

4.1.1 Design Goals and Contributions

Reliability is a central concern for wireless sensor networks, and developing fault-tolerant distributed applications is challenging, especially for those with professions other than software engineering, i.e., most people with a need for distributed sensing. Our goal is to combine high-level programming languages and automatic fault-aware code transformation techniques to empower novice programmers to develop sensor networks that can operate reliably, potentially in harsh environments. Our work is based on three insights. (1) Novice programmers tend to assume a fault-free system during programming, e.g., corner cases are often ignored by beginners. (2) Application experts mostly care about application-level performance; they should be informed about the impact of errors on the end products of an application, but reporting every detail about low-level sensor network faults may impose great burden with little value. (3) The knowledge of application experts about expected behaviors and environmental conditions can be used to allow more effective fault detection and correction.

We have designed, implemented, and evaluated a system, called FACTS (Fault-Aware Code Transformation for Sensor networks), to simplify programming faulty sensor networks. FACTS hides faults from programmers but indicates the impact of low-level faults on application outputs, i.e., the end results of data processing expressions in the application specification. We implement FACTS by extending an existing high-level programming language for sensor networks. The current design of FACTS focuses on data-acquisition applications and sensor data faults. Programmers provide specifications of application logic as well as expected environmental conditions. The compiler automatically generates fault-aware code to which fault detection, error correction, and error estimation functionalities have been added. During network operation, sensor faults are detected by identifying sensor readings that fall outside of application-specific ranges. In case of sensor faults, the ranges of actual data values are estimated using temporal and spatial correlation. The ranges of end results produced via potentially complex expressions are then computed.

Our work makes three main contributions.

1. We describe an approach to simplify programming of potentially faulty sensor networks by automatically generating code for fault detection, error correction, and error estimation.

2. We develop an error estimation technique to calculate the error bounds for application data as a result of faults.

3. We implement this approach in a real system by modifying the compiler and runtime library for a high-level sensor network programming language.

We evaluate the overhead of our system on code size, memory use, and improvement in end result accuracy using a small-scale testbed and simulation of larger-scale networks. The average code size overhead is 15% and the average memory overhead is 3.6%. The resulting intervals produced by FACTS always contain the actual value, while the fault-unaware program can produce substantial errors.

Note that our work does not focus on designing new methods for fault detection or fault tolerance, though it builds upon and benefits from existing fault tolerance techniques.

4.1.2 Related Work

Although the high-level programming languages [82, 95, 17, 98, 9] have reduced programming complexity compared to node-level programming languages [39], none provide support for fault detection and error correction. Wireless sensor network components commonly experience faults [134, 63, 49] so using fault detection and tolerance techniques has great value. Even if programmers are willing to deal with greatly increased implementation complexity, some macroprogramming languages provide no programmer access to node-level communication primitives, making it intractable for the programmer to implement fault detection and correction techniques. To the best of our knowledge, only one sensor network language explicitly supports fault tolerance [47]. It provides declarative annotations to specify checkpointing recovery strategies. In contrast, our system (FACTS) does not require programmers to explicitly deal with faults, making it accessible even to sensing application experts with limited programming experience.

Researchers have identified and classified various types of faults in sensor networks and proposed numerous approaches for fault detection [60], tolerance [26, 76], diagnosis [77, 112], and recovery [99, 47]. These papers concentrate on minimizing the impact of faults on system performance and availability. We intend to use them to build useful sensor network design tools that automatically use these techniques without requiring programmers to understand the details of, or manually implement, them. We also propose an error estimation technique to provide application experts with a more accurate and informative view of data gathered from a network.

4.2 Fault-Aware Code Transformation for Sensor Networks (FACTS)

We now describe the FACTS architecture.

4.2.1 FACTS System Architecture

The purpose of FACTS is to shift responsibility for the mechanical aspects of fault management from programmers to the programming language, compiler, and run-time libraries. In this paper, we focus on data acquisition applications. The left hand side

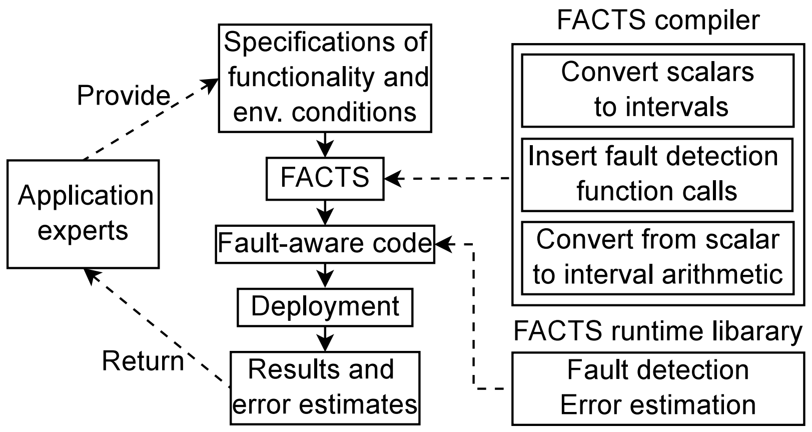

FACTS compiler
Specifications of functionality and env. conditions
Provide
Convert scalars to intervals
FACTS
Insert fault detection function calls
Application experts
Fault-aware code
Convert from scalar to interval arithmetic
Deployment
FACTS runtime libarary
Return
Results and error estimates
Fault detection
Error estimation

Figure 4.1: Separation of reliability concerns.

of Figure 4.1 illustrates how application experts use our system. An application expert specifies application functionality and expected environmental conditions. FACTS uses this information to generate an implementation that is capable of fault detection, fault recovery, and error estimation. The application expert then deploys a network running the generated code. FACTS indicates the application-level impact of faults, i.e., the error range for the end results of potentially complex data processing expressions, while hiding component-level implementation and fault details from the application expert.

We argue for focusing on the high-level implications of faults for three reasons: (1) data quality is important to application experts; (2) detailed information about low-level faults (e.g., which sensor readings are out of range) is generally overwhelming instead of informative to application experts; and (3) providing detailed information about faults is costly in terms of energy consumption and it is not necessary except for the purpose of debugging and system diagnosis.

The right hand side of Figure 4.1 shows the system components and their purposes. The original compiler generates node-level code that implements sensing, data transmission, and data aggregation algorithms. FACTS provides a runtime library to detect faults and estimate errors. The FACTS compiler modifies and augments the original compiler

Table 4.1: Example of Fault Correction

	Node 1	Node 2	Node 3	Average
True value (°C)	10	12	14	12
Sensor reading (°C)	10	12	0	7.3
Corrected reading (°C)	10	12	[10, 16]	[10.7, 12.7]

in the following ways to generate fault-aware code. (1) It changes the types of some variables in the program to include extra information about error estimates. (2) It transforms arithmetic expressions to interval arithmetic expressions so the implications of faults can be propagated to the end results. (3) It inserts calls to fault detection and error estimation functions.

We now use an example to demonstrate the key ideas of our approach. Consider the application that monitors redwood tree microclimates [137]. Biologists deploy sensor nodes on a redwood tree to gather temperature and humidity data. The application periodically samples temperature and humidity, averages readings from nodes at similar heights, and sends the results to a base station. Assume at one height, there are three nodes with identifiers 1, 2, and 3. Table 4.1 shows an example of data gathered during one sampling cycle. The second row shows the ground truth values for each node. The third row shows the sensor readings. Node 3 is faulty: the fault results in an erroneous sensor reading of 0 °C. Without any fault tolerance mechanisms, the average of the three values in the second row, 7.3 °C, is returned to the user. Unfortunately, the user is unaware that 7.3 °C is an erroneous result that underestimates the average temperature by 4.7 °C.

With FACTS, the expert designing the application provides some information about the environment in a simple format. For example, the expected temperature range is 10–30 °C. The code generated by FACTS uses this information to detect the fault at node 3. Instead of using the incorrect value of 0 °C, it indicates that the value is in the range 10–16 °C based on historical readings and readings from other nearby nodes. FACTS then propagates this interval through the expressions to produce the value of interest for the network (i.e., the average), indicating that it is in the range 10.7–12.7 °C. The user is made aware of the system-level implications of the low-level fault. This error information can be further used by application experts during their data analysis and help them draw more accurate scientific conclusions. The actual techniques used in FACTS are more sophisticated than those considered in this explanatory example. For example, FACTS considers the influence of spatial and temporal correlation as well as the impact of expressions predicated on faulty variables.

Our approach has the following features.

1. Application experts do not need to understand the intricacies of sensor network faults or explicitly manage them.

2. The domain knowledge of application experts is used to allow fault detection and error estimation, without imposing much additional specification burden.

3. System-level error bounds are provided to application experts to allow more thorough understanding of data.

4.2.2 Specification of Environmental Conditions

Application experts' knowledge of environmental conditions can be used for two purposes: detecting sensor faults and correcting for faulty sensor readings. Sensed data characteristics can be determined based on sensor specifications and environmental conditions such as data value range, temporal gradient (change in value per time unit), temporal correlation, spatial variance (change in value per distance unit), and spatial correlation. In this work we use range, maximum temporal gradient, and maximum spatial gradient to describe the environmental conditions; however, these concepts can be extended to use other parameters. We extend WASP programming language to let programmers specify an expected range and maximum temporal/spatial gradient for each environmental parameter. After the programmer provides the application specification, a list of relevant physical parameters is extracted to produce a template for the programmer to input information about their expected behaviors. The programmer need only read the template and enter a few numbers.

4.2.3 Fault Detection and Sensor Error Estimation

In this work, we focus on methods that can be implemented efficiently in software and detect a commonly occurring class of faults. Although the proposed error estimation technique will work with any hardware or software fault detection mechanism, we use the following detection criteria in our FACTS system prototype: (1) are the sensor data within the expected range? and (2) are the environmental conditions within the operating range of the sensors?

It is common for faulty sensor nodes to produce abnormal readings. For example, developers of a habitat monitoring network observed abnormally large or small sensor readings (light, temperature, and humidity) when water penetrated the enclosure of the sensor node and affected the power supply [134]. Developers of a redwood tree macroclimate monitoring network associated out-of-range sensor readings with node faults caused by a drop in battery voltage [137]. Such faults can be detected via range checking.

As sensors cannot work properly in certain environmental conditions, sensor faults can also be detected by checking whether the current environmental conditions are within the sensor's operating range. If either requirement is violated, the sensor reading is deemed incorrect. Consider an application that gathers light level readings using TelosB sensor nodes. The S1087 light sensor on TelosB nodes has an operating temperature range of -

10–60 °C. Both the light and temperature sensor readings are checked to detect light sensor data faults. A fault is likely to occur if either the light sensor reading is out of the expected range or the temperature sensor reading is out of the -10–60 °C range. Note that when an undetected fault occurs in the temperature sensor, we may (conservatively but sometimes mistakenly) deem the light sensor to be faulty. Faults in sensors on the same node may be correlated because they share many hardware and software components; the developers of the habitat monitoring sensor network observed this correlation [134]. Therefore, the false positive rate due to faults in the sensor monitoring the operating environment is likely to be lower than would be the case in the absence of sensor fault correlation.

Local error estimation is used to indicate the intervals of actual data elements and expressions when faults are detected. Faulty sensor readings are estimated based on bounds on environmental parameters and their spatial and temporal gradients. Data gathered from a sensor network often change gradually with time and location. Temporal and spatial variations can be bounded for many applications. We use such bounds to replace erroneous values with ranges. For example, given a maximum temporal gradient for temperature of 1 °C per minute, the range of a faulty temperature reading can be estimated as 19–21 °C if the most recent correct reading of 20 °C was taken one minute ago. In other words, in case of an erroneous reading, the possible temperature range is estimated based on other data. The FACTS compiler creates data buffers to store historical data. The buffer size is determined based on the user-specified bounds and sampling periods. For example, if the temporal variation of temperature is at most 5 °C over one hour and the temperature sampling period is 10 minutes, then a buffer of size 6 is used.

A bound on spatial gradient indicates the maximum change per meter. Error estimation using spatial gradients requires knowledge of distances between sensor nodes. If node locations are known at design time, the locations of nearby nodes can be stored in a table

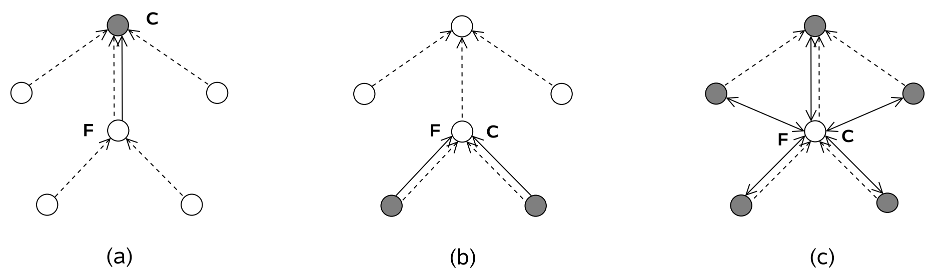


Figure 4.2: Design options for error estimation based on spatial data.

and used for error estimation at runtime. If node locations are unknown until deployment, distances between nodes can be estimated using node localization algorithms [117].

Error estimation using spatial gradients requires data from other nodes and may therefore introduce communication overhead. The locations where the implications of faults are estimated and the amount of spatial data used impact energy overhead and the tightness of the resulting error bounds. Figure 4.2 shows examples of three design options for a network composed of six nodes. A dotted arrow represents a communication link in the routing tree, originating from a child node and ending at a parent node. **F** indicates where a fault occurs and **C** indicates where the error resulting from this faulty reading is estimated. The center node has a faulty sensor reading. To simplify explanation, we ignore error estimation based on temporal changes in this example. Sensor data from the shaded nodes are used to estimate the interval for incorrect readings gathered by the faulty sensor. A solid arrow indicates the links on which the corresponding design option produces communication overhead.

In design (a) (in Figure 4.2), the parent node estimates the value interval for the faulty node by based on the parent node's sensor value. In design (b), the value interval at the faulty node is estimated using its children's readings. In design (c), the value interval at the faulty node is estimated using all its neighbors' readings. Design (a) uses only one neighboring node (parent node) for estimation, design (b) uses all children nodes, and design

(c) uses all neighboring nodes. Design (a) imposes communication overhead on the link from a faulty node to its parent, design (b) requires every node to always send its own raw sensor readings to its parent, design (c) requires the faulty node to broadcast requests to which its neighbor nodes reply with their sensor readings. The more information used in estimating an interval, the tighter the bound is; design (a) provides the loosest bound with the lowest communication overhead and design (c) provides the tightest bound with the highest communication overhead. We choose design (b) in the FACTS system implementation. This option requires the least modification to the network protocol, and supports the use of multiple spatial readings for error estimation. Specifically, each node sends the aggregated results of the subtree it is the root for and its own raw sensor reading.

4.2.4 Error Propagation

The WASP programming language supports node-level data processing functions and network-level aggregation functions. FACTS computes the errors in expression results based on the sensor readings they depend on. Specifically, FACTS returns estimated ranges associated with each requested datum, i.e., every data element in the `COLLECT` statement for network-level data gathering and aggregation in a WASP program. As described in Section 4.2.3, faulty sensor readings are replaced with estimated ranges. The errors are then propagated to final results using interval arithmetic. Error estimation for network-level aggregation results can occur either in the network or at the base station. The former approach aggregates correct and faulty variables in the network and estimates the associated error. The latter approach aggregates only correct variables in the network and forwards faulty variables to the base station, where the error of the final results are computed. We adopt the former approach in FACTS because it implies smaller data transmission overhead and scales with network size and fault rate.

Errors caused by faulty sensor readings can propagate to end results via mathematical operations such as addition. The error estimation problem can be defined as follows. Given $y = f(x_1, x_2, \cdots, x_n)$ and the range of each x_i, estimate the range of y. Each x_i represents a potentially erroneous variable. y represents the returned result. This can be solved with interval arithmetic [90], in which arithmetic operations are applied to operand intervals to calculate result intervals. Interval arithmetic has been applied to rounding error estimation and circuit timing analysis. In contrast to these uses, maintaining low overhead is more important for our (on-line) application because error estimation may execute on energy- and time-constrained sensor nodes. Fortunately, for the built-in functions supported by the WASP programming language, it is easy to find the range of an output given ranges of inputs. For example, the frequently used aggregation functions such as `MAX`, `MIN`, and `AVG` are all monotonic, allowing the extremes of an output to be computed directly by applying the operation on extremes of inputs. The mathematical expressions and functions used in the majority of published wireless sensor network deployments can also be efficiently computed following interval arithmetic rules.

Errors can also propagate to end results via their influence on control flow. When a faulty variable is used in a predicate expression and its estimated range spans the predicate threshold, the range of the result is computed by combining the ranges that would result from either branch. For non-aggregating data collection applications, the predicates determine whether data should be sent to the base station. For applications with in-network spatial data aggregation, the predicates determine whether data should be included in the network-level aggregation operations. For example, `COLLECT AVG(y) WHERE x > 100` requires the y variable of a particular node to be included in the averaging operation if that node's x value is above 100. When a fault results in the interval for x spanning 100, the range of interval `AVG(y)` (the average of all node y values) should span the results calculated with and without including y from the faulty node. The error estimation problem can more formally be defined as follows.

Given that $y = f(x_1, x_2, \cdots, x_n)$ where f is an aggregation function for which x_i may or may not be included in the argument list and n is number of variables, compute the range of y. The range of y can be naïvely obtained by computing the ranges for the 2^n cases separately and calculating their union. The computational complexity may be acceptable for a sparse network since n is bounded by the maximum number of immediate children nodes. Fortunately, for the aggregation operations commonly used in wireless sensor network deployments, the computational complexity is less than $\mathcal{O}(n \log n)$. For `MAX` and `MIN` operations, including one more argument only monotonically affects the upper or lower bound of the result. Therefore, the range of the result can be easily calculated by iteratively considering each of the n variables. For `AVG`, adding one more variable may increase or decrease both lower and upper bounds. However, the range of the result can still be computed in $\mathcal{O}(n \log n)$ time. For example, to get the lower bound of the `AVG` result, first order the lower bounds of the intervals associated with the nodes that may meet the selection requirements (but are not certain to meet them) in increasing order. Incrementally scan the ordered list, add each value to the set of values to average, and recompute the result until the local minimum for the result is reached. To get the upper bound of the `AVG` result, use a similar technique, but instead scan the upper bounds of the intervals in decreasing order. Consider the expression `COLLECT AVG(y) WHERE x > 100` as an example. Assume a network of five nodes. Their x ranges are [120], [90, 110], [80, 120], [83, 102], and [130]. Their y values are 2, 4, 6, 3, and 8. To compute the upper bound of `AVG(y)`, we order the y values except 2 and 8 (they must be involved in computing the average) in descending order. The average of 2 and 8 is 5. After including 6, the average becomes 5.3. After including 4, the average becomes 5. Therefore, the upper bound for

`AVG(y)` is 5.3.

4.2.5 Automated Code Transformation

We now describe a software implementation to support automatic online fault detection and error estimation. The following steps will be used to generate fault-aware code. (1) Replace sensor readings and the variables that depend upon them with tuples containing two variables of the same type. (2) Insert calls to fault detection functions after sensor readings are obtained. The fault detection methods have been described in Section 4.2.3. (3) Insert calls to temporal gradient-based error estimation functions after error detection function calls to calculate ranges for faulty variables. (4) Insert calls to spatial gradient based error estimation functions before a node aggregates its received data. (5) Convert mathematical expressions involving possibly faulty variables to interval arithmetic operations. For example, $z = x+y$ is converted to $z.low = x.low+y.low$; $z.high = x.high+y.high$.

4.3 Experimental Evaluation

We evaluated the accuracy of the value estimates provided by FACTS, as well as its impact on code size and memory use. Our evaluation uses a small-scale experimental hardware testbed and simulations of a larger-scale network composed of 74 sensor nodes with real-world data traces. This section describes the experimental setup and the results.

4.3.1 Prototype Evaluation

We implemented a prototype of the proposed system and tested it in a small-scale sensor network consisting of four TelosB nodes. Each node samples temperature every 2 s. The average across all nodes is returned to the base station. The results contain a tuple for each sampling cycle indicating the upper and lower bounds on the average temperature. Figure 4.3 displays the temperature upper and lower bounds as functions of time.

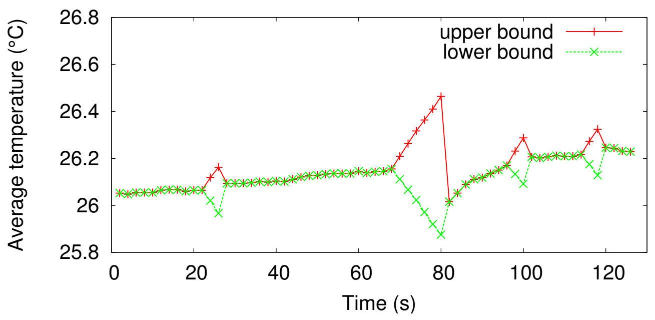


Figure 4.3: Temperature interval as a function of time.

Table 4.2: Fault-Aware and Unaware Implementations

	Code size (B)			Memory usage (B)		
	App.1	App.2	App.3	App.1	App.2	App.3
Fault-unaware	32,556	33,060	27,722	2,130	2,134	2,038
Fault-aware	37,358	37,740	32,088	2,212	2,224	2,096
Overhead (%)	14.7	14.2	15.7	3.8	4.2	2.7

We injected intermittent sensor faults by shorting the terminals of the thermal sensor to produce readings of -39.6 °C (from a 0 V analog-to-digital converter input), e.g., at 22 s. In the absence of faults, the upper and lower bounds in Figure 4.3 are identical. The estimated bounds become looser over time when the intermittent fault persists, due to the use of temporal correlation to calculate the temperature interval. This section serves primarily to demonstrate that a functioning prototype of the FACTS system has been implemented, and explain its operation.

4.3.2 Evaluation of Code Size and Memory Use Overhead

To evaluate the impact of using FACTS on code size and memory requirements, we compared the code generated with the original WASP compiler and the extended FACTS compiler. Table 4.2 shows the code size and memory use for the three representative examples based on deployed sensor network applications [130]. Application 1 periodically gathers temperature and light data. Application 2 periodically samples light and averages

Table 4.3: Lines of Code for Fault-Aware and Unaware Implementations

	High-level specification			NesC code		
	App.1	App.2	App.3	App.1	App.2	App.3
Fault-unaware	6	7	7	489	495	484
Fault-aware	12	10	10	621	585	545

data among nodes at similar heights. Application 3 periodically samples temperature and sends data only when the increase in temperature exceeds a threshold. The average code size overhead across the three applications is 15% and the average memory overhead is 3.6%.

We compare the lines of code for the high-level specification input to FACTS as well as the generated node-level code to give some evidence of its impact on programming complexity. The results are shown in Table 4.3. The applications are the same as those used for code size and memory use estimation. FACTS only requires three to six additional lines of code in the high-level specification, depending on how many physical parameters are sensed. Note that the syntax of the FACTS specifications is at least as simple as that for functionality. Therefore, we argue that the programming complexity only increases slightly. Given that researchers have previously demonstrated via user studies that novice programmers can use WASP correctly and efficiently [130], and the additional specifications required by FACTS have low complexity and length, we believe that the FACTS system will remain accessible to novice programmers. In contrast, the low-level NesC code (excluding library code) increases in length by 61, 90, or 132 lines of code depending on application. This implies that the extra programming efforts required to manually and explicitly handle sensor faults is potentially high. With FACTS, the increased implementation complexity is not exposed to programmers.

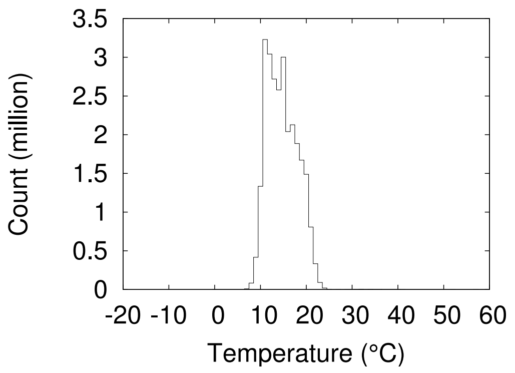


Figure 4.4: Temperature histogram.

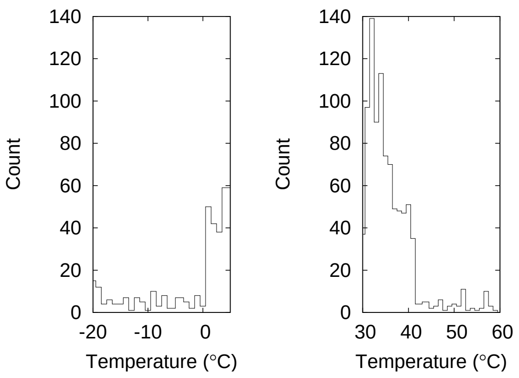


Figure 4.5: Outlier histograms.

4.3.3 Simulation of Large-Scale Network to Evaluate Impact of Varying Fault Rates

Simulation environment: We use the SIDnet-SWANS simulator [43] and temperature measurement time series from a real network deployment [13] to model a network of 74 nodes that sample temperature every 29.3 seconds and aggregate data in the network. We assume two aggregation expressions: average and minimum.

Environmental data generation: We use the data from the LUCE deployment at the EPFL campus [13] to provide environmental data for our simulation. The LUCE deployment contains 97 weather stations that span a 500 m×300 m area and ran for 6 months. We take the following steps to generate fault-free data traces from the original data set. (1) It is important that faults be rare in our input data so that we can determine the actual ground truth data values with which our fault correction system ranges and estimates will be compared. We identified a time interval in which the data drop rate from most of the nodes is small and eliminated from consideration 23 nodes that have high drop rates in that time interval. We used a one-hour trace containing 9,028 data samples from 74 nodes. (2) The original data set has a period of 29.3 s with small jitter. We parse the data to produce synchronized periodic time series. Multiple samples associated with the same period are averaged, while periods without data are recovered by selecting from the valid data value distribution for the application. Combined, these faults only affected 3.7% of the time series data. (3) We determine the lower and upper bounds based on the histogram of temperature data from the 74 nodes. The histogram is shown in Figure 4.4, where 99.4% of the data are in the 5–30 °C range. Inspection indicates that data outside this range are associated with spikes in the time series. We treat data outside this range as outliers. (4) We analyze the temporal and spatial correlation ignoring outliers and compute the bounds on temporal and spatial gradients. The results are 3 °C per 29.3 s and 5 °C per 50 m. (5) We replace faulty and missing data (only 3.7% of the original data traces) with values that are generated based on spatial and temporal correlation. The resulting data set complies with the bounds on data range, temporal gradient, and spatial gradient.

Fault injection during evaluation: We model sensor transient faults using Poisson processes, primarily because many transient fault processes are memory-less. This influences only our simulation results, not the design of the FACTS system, which can handle fault processes with arbitrary temporal density functions. We use independent but equal-rate fault processes for different sensor nodes. Faulty sensor readings are generated by sampling from the set of outliers extracted from the original data set; this was done so that the simulated and actual faulty data would have the same distribution, which is shown in Figure 4.5. We run simulations with multiple fault rates, ranging from 0.1 to 0.5 per minute, to study the impact of fault rate on accuracy. The tested fault rates are selected based on a survey on sensor network data faults by Ramanathan et al. [112]. The sensor fault durations in the original data are generally less than 29.3 s (one sampling cycle), supporting the injection of transient faults. For each fault rate, we run 5 simulations with different random seeds and average the results.

Results: Figure 4.6 shows an example simulated time series for a fault rate of 0.1 per minute. The shaded area shows the value intervals produced by FACTS. The curve

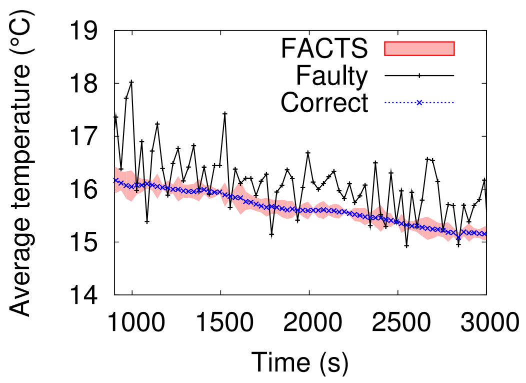


Figure 4.6: Results with and without FACTS.

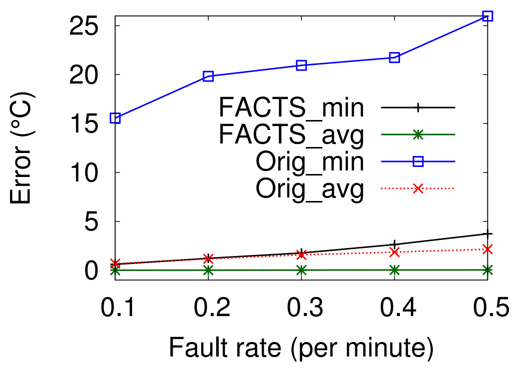


Figure 4.7: Dependence of error (in °C) on fault rate.

inside it shows ground truth results from the original time series. The figure shows that the original fault-unaware program can produce substantial errors (0.7 °C on average and 2.7 °C maximum) and that the intervals produced by FACTS always contain the actual value.

If the midpoints of the intervals produced by FACTS are used as value estimates, the error relative to ground truth values can be computed. Figure 4.7 shows aggregate error for simulation runs with different fault rates. The root mean square errors relative to the ground truth data are computed for the fault-aware and fault-unaware programs. FACTS_min and FACTS_avg represent the results for the minimum expression application and the average expression application. Orig_min and Orig_avg represent the results for the fault-unaware versions of these applications. FACTS results have an average error of 0.02 °C and fault-unaware results have an average error of 3.86 °C. The worst-case errors are 5.95 °C and 36.40 °C for FACTS and the fault-unaware systems.

4.4 Conclusions

We have described FACTS, a system to simplify fault detection and correction in wireless sensor networks that is designed to be accessible to application experts who may not be expert programmers. FACTS uses easily specified domain-specific expert knowledge to support the on-line detection of some classes of sensor faults and appropriately adjust expression intervals to make the system-level impact of faults clear to sensor network users. We implemented FACTS by extending the WASP sensor network language, compiler, and run-time system. A small-scale hardware testbed and simulations of a 74-node network using real-world sensor data show that FACTS substantially increases estimation accuracy and imposes little overhead compared to fault-unaware programs. Our work focuses on a common type of fault in sensor networks and an error handling method that is useful for data-centric applications. To be applied to a larger domain of sensor network applications, other types of faults and error handling methods need to be considered.

CHAPTER V

Memory Expansion for MMU-Less Embedded Systems

Random access memory (RAM) is tightly-constrained sensor network nodes. The most widely-used sensor network nodes have only 4–10 KB of RAM and do not contain memory management units (MMUs). It is difficult to implement complex applications under such tight memory constraints. Nonetheless, price and power consumption constraints make it unlikely that increases in RAM in these systems will keep pace with the increasing memory requirements of applications.

In this chapter, we present automated compile-time and run-time techniques to increase the amount of usable memory in sensor nodes. The proposed techniques do not increase hardware cost, and require few or no changes to existing applications. We have developed run-time library routines and compiler transformations to control and optimize the automatic migration of application data between compressed and uncompressed memory regions, as well as a fast compression algorithm well suited to this application. These techniques were experimentally evaluated on Crossbow TelosB sensor network nodes running a number of data-collection and signal-processing applications. Our results indicate that available memory can be increased by up to 50% with less than 10% performance degradation for most benchmarks. Our approach can be applied to other MMU-less embedded systems.

The rest of this chapter is organized as follows. Section 5.2 summarizes related work and contributions. Section 5.3 provides a motivational scenario that illustrates the importance of the proposed technique. Section 5.4 describes the library and compiler techniques, optimization schemes, as well as the compression and decompression algorithms designed to automatically increase usable memory in sensor network nodes. Section 5.5 presents the experimental set-up, describes the workloads, and discusses the experimental results in detail. Finally, Section 5.6 concludes the article.

5.1 Introduction

Low-power, inexpensive embedded systems are of great importance in applications ranging from wireless sensor networks to consumer electronics. In these systems, processing power and physical memory are tightly limited due to constraints on cost, size, and power consumption. Moreover, many microcontrollers lack memory management units (MMUs). Sensor network nodes have tight price and power constraints. Although the proposed techniques may be used in any memory-constrained embedded system without an MMU, this article will focus on using them to increase usable memory in sensor network nodes with no changes to hardware and with no or minimal changes to applications.

Many recent ideas for improving the communication, security, and in-network processing capabilities of sensor networks rely on sophisticated routing [56], encryption [38], query processing [41], and signal processing [72] algorithms implemented on sensor network nodes. However, sensor network nodes have tight memory constraints. For example, the popular Crossbow MICA2, MICAz, and TelosB sensor network nodes have 4 KB or 10 KB of RAM, a substantial portion of which is consumed by the operating system (OS), e.g., TinyOS [40] or MANTIS OS [3]. Tight constraints on the cost and power consumption of sensor network nodes make it unlikely for the size of physical RAM to keep pace with the demands of increasingly sophisticated in-network processing algorithms.

In order to reduce cost, sensor network nodes typically avoid the use of dedicated dynamic random access memory (DRAM) integrated circuits; in extremely low price, low power embedded systems, RAM is typically on the same die as the processor. Unfortunately, it is not economical to fabricate the deep trench capacitors used for high-density DRAM with the same process as processor logic. As a result, static random access memory (SRAM) is used in sensor network nodes. Unlike DRAM, SRAM generally requires six transistors per bit and has high power consumption. Increasing the amount of physical memory in sensor network nodes would increase die size, cost, and power consumption. Some researchers have proposed addressing memory constraints using hardware techniques such as compression units inserted between memory and processor. However, such hardware implementations typically have difficulty adapting to the characteristics of different application data. Moreover, they would increase the price of sensor network nodes by requiring either additional integrated circuit packages or microcontroller redesign. Barring new technologies that allow inexpensive, high-density, low-power, high-performance RAM to be fabricated on the same integrated circuits as logic, sensor network applications will continue to face strict constraints on RAM in the future.

Software techniques that use data compression to increase usable memory have advantages over hardware techniques. They do not require processor or printed circuit board redesign and they allow the selection and modification of compression algorithms, permitting good performance and compression ratio (compressed data size divided by original data size) for the target application. However, software techniques that require the re-design of applications are unlikely to be used by anybody but embedded systems programming experts. Unfortunately, most sensor network application experts are not embedded system programming experts. If memory expansion technologies are to be widely deployed, they should not require changes to hardware and should require minimal or no changes to applications. Motivated by the above observations, we propose a new software-based on-line memory expansion technique, named MEMMU, for use in wireless sensor networks.

5.2 Related Work and Contributions

The proposed library and compiler techniques to increase usable memory build upon work in the areas of on-line data compression, wireless sensor networks, and high-performance data compression algorithms.

Software Virtual Memory Management for MMU-Less Embedded Systems Choudhuri and Givargis [25] proposed a software virtual memory implementation for MMU-less embedded systems based on an application level virtual memory library and a virtual memory aware assembler. They assume secondary storage, e.g., EEPROM or Flash, is present in the system. Their technique automatically manages data migration between RAM and secondary storage to give applications access to more memory than provided by physical RAM. However, since accessing secondary storage is significantly slower than accessing RAM, the performance penalty of this approach can be very high for some applications. In contrast, MEMMU requires no secondary storage. In addition, its performance and power consumption penalties have been minimized via compile-time and runtime optimization techniques.

Hardware-Based Code and Data Compression in Embedded Systems A number of previous approaches incorporated compression into the memory hierarchy for different goals. Main memory compression techniques [138] insert a hardware compression/decompression unit between cache and RAM. Data are stored uncompressed in cache, and are compressed on-line when transferred to memory. Main memory compression techniques are used to improve the system performance by providing virtually larger memory. Code compression techniques [67] store instructions in compressed format in ROM and decompress them during execution. Compression is usually performed off-line and can be slow, while decompression is done during execution, usually by special hardware, and must be very fast. Code compression techniques are often used to save space in ROM for embedded systems with tight resource constraints.

Software-Based Memory Compression Compressed caching [29,143] introduces a software cache to the virtual memory system. This cache uses part of the memory to store data in compressed format. Swap compression [139] compresses swapped pages and stores them in a memory region that acts as a cache between memory and disk. The primary objective of both techniques is to improve system performance by decreasing the number of page faults that must be serviced by hard disks. Both techniques require backing store, i.e., a hard disk, when the compressed cache is filled up. In contrast, MEMMU does not rely on any backing store.

CRAMES [36] is an OS controlled, on-line memory compression framework designed for disk-less embedded systems. It takes advantage of the OS virtual memory infrastructure and stores least recently used (LRU) pages in compressed format in physical RAM. CRAMES dynamically adjusts the size of the compressed memory area, protecting applications capable of running without it from performance or energy consumption penalties. Although CRAMES does not require any special hardware for compression/decompression, it does require an MMU. In contrast, MEMMU requires no MMU. In fact, MEMMU implements software memory management via its compile-time and runtime techniques and uses numerous optimizations to maintain performance. This capability is necessary for most sensor network nodes and low-cost embedded processors because the majority do not have MMUs.

Biswas et al. [18] described a memory reuse method that relies upon static liveness analysis. It compresses live globals in place and grows the stack or heap into the freed region when they overflow. Their work aims at improving system reliability by resolving runtime memory shortage errors as a consequence of the difficulty in predicting the size requirement of dynamic memory objects such as stack and heap. In contrast, MEMMU tries to solve a different problem: permitting system operation when the lower bound on memory requirements already surpass physical memory. Therefore, MEMMU has a much bigger memory expansion ratio.

Garbage collection is another approach to reduce memory footprint by reclaiming memory occupied by unreachable objects. Researchers have designed garbage collectors with small code size and memory overhead for embedded systems [6]. Their approach is complementary to MEMMU. Garbage collection based techniques are useful for programs using dynamic memory allocation. MEMMU focuses on programs using static memory allocation in which memory objects can be alive during the whole lifetime of the program.

Cooprider and Regehr [28] proposed an RAM compression technique that targets data elements that have values limited to small sets, which are determined using compile-time analysis. In contrast, MEMMU uses on-line compression of data based on access patterns that are hard to determine at compile time. As a result, MEMMU can be applied to sensor data, generally permitting greater increases in usable memory. Note that Cooprider's and Regehr's technique, and MEMMU, are complementary; they compress different structures and do not significantly interfere with each other.

Compression for Reducing Communication in Sensor Networks

In many sensor network applications, sensor nodes in the network must frequently communicate with each other or with a central server. Sensor nodes have limited power sources and wireless communication accelerates battery depletion [109]. In-network data aggregation [83,46] and data reduction via wavelet transform or distributed regression [50, 97] can significantly reduce the volume of data communicated. However, these techniques are lossy, limiting their application. Recently, researchers have proposed to reduce the amount of data communication via compression [105, 110] in order to reduce radio energy consumption. Our work differs from theirs in that MEMMU focuses on automated memory compression for functionality improvement instead of communication reduction.

Software-Based Memory Compression Algorithms

LZO [100] is a very fast general-purpose compression algorithm that works well on many in-RAM data. However, the memory requirement of LZO is at least 8 KB, far exceeding the available memory of many low-end embedded systems and sensor nodes. Rizzo et al. [114] proposed a software-based algorithm that compresses in-RAM data by only exploiting the high frequency of zero-valued data. This algorithm trades off degraded compression ratio for improved performance. Wilson et al. [143] presented a software-based algorithm called WKdm that uses a small dictionary of recently-seen words and attempts to fully or partially match incoming data with an entry in the dictionary. Yang et al. [37] designed a software-based memory compression algorithm for embedded systems named pattern-based partial match (PBPM). This algorithm explores frequent patterns that occur within each word of memory and exploits similarities among words.

Many software-based memory compression algorithms are not appropriate for use on sensor network nodes due to large memory requirements or poor performance. For those with sufficiently low overhead, we found none that provides a satisfactory compression ratio for sensor data. The main reasons for this follow:

1. Zero words are rare in many forms of sensor data.

2. Many forms of sensor data change gradually with time. As a result, adjacent data elements are often similar in magnitude but have very different bit patterns. Therefore, conventional dictionary-based compression does not work well. We evaluated a partial dictionary match algorithm [37] in this application but the compression ratio was much worse than delta compression. The partial dictionary match achieved an 86% compression ratio for trace data while the proposed delta compression algorithm achieved a 50% compression ratio. We suspect that part of the cause for the poor performance of the dictionary-based algorithm was the high relative penalty for storing dictionary indices when 16-bit words are used; the algorithm performs well in another application in which 32-bit words are used.

3. The block size used in compression is often restricted in low-cost MMU-less devices, as we will explain later.

We propose a memory compression algorithm that operates with very high performance on the 16-bit data generally found in the memory of MICAz and TelosB sensor network nodes. The average compression ratio for various types of sensor data is approximately 50%.

Contributions The proposed memory expansion technique, MEMMU, expands the memory available to applications by selectively compressing data that reside in physical memory. MEMMU uses compile-time transformations and runtime library support to automatically manage on-line migration of data between compressed and uncompressed memory regions in sensor network nodes.

MEMMU essentially provides a compressed RAM-resident virtual memory system that is implemented completely in software via compiler transformations and library routines. Its use requires no hardware MMU, and requires few or no manual changes to application software.

Our work makes four main contributions.

1. It provides application developers with access to more usable RAM and requires no or minor changes to application code and no changes to hardware.

2. It does not require the presence of an MMU and has other design features that enable its use in sensor network nodes with extremely tight memory and performance constraints.

3. It has been optimized to minimize impact on performance and power consumption; experimental results indicate that in many applications, such as data sampling and audio signal correlation computation, its performance overhead is less than 10%.

4. We have released MEMMU for free academic and non-profit use [87].

MEMMU was evaluated on TelosB wireless sensor network nodes. The TelosB is an MMU-less, low-power, wireless module with integrated sensors, radio, antenna, and an 8 MHz Texas Instruments MSP430 microcontroller. The TelosB has 10 KB RAM and typically runs TinyOS.

5.3 Motivating Scenario

In this section, we describe a motivating scenario that illustrates the purpose and operation of MEMMU. Consider an application in which individual sensor nodes react to particular events, e.g., low-frequency vibration, by triggering high rate audio data sampling. After the sampling is complete, data are filtered and statistics, e.g., variance and mean, are computed and transferred to an observer node. If the raw data are of interest to the observer node, they are requested and transmitted through the network. In existing sensing architectures, the size of the data buffer is tightly constrained. For example, on a Crossbow TelosB sensor node a maximum of 9.5 KB RAM is available for buffering. Moreover, sampling rate and duration cannot be increased without redesigning the sensor node hardware or increasing the complexity of application implementation. If, instead, the automated data compression technique proposed in this article is used, portions of sampled data will be automatically compressed whenever they would otherwise exceed physical memory. During filtering, e.g., convolution, data are automatically decompressed and recompressed to trade off performance and usable memory. Commonly-accessed data are cached in uncompressed format to maintain good performance. This is achieved without changes to hardware and with no or minimal changes to application code. To the application designer, it appears as if the sensor network node has more memory than is physically presented.

Many wireless sensor networks use a store-and-forward technique to distribute information. Therefore, the local memory of a node is used as a shared resource to handle multiple messages traveling along different routes. In order to avoid losing data during communication, a node must generally store already-sent data until it receives an acknowledgment. As a result, the buffer can easily be filled when the communication rate is high, leading to message loss or even network deadlock. With MEMMU, usable local memory can be increased thus reducing the probability of data loss.

5.4 Memory Expansion on Embedded Systems Without MMUs

This section describes the design MEMMU, our technique for *Memory Expansion on embedded systems without MMUs*. The main goal of MEMMU is to provide application designers with access to more usable RAM than is physically available in MMU-less embedded systems without requiring changes to hardware and with minimal or no changes to applications. We achieve this goal via on-line compression and decompression of in-RAM data. In order to maximize the increase in usable RAM and minimize the performance and energy penalties resulting from the technique, it is necessary to solve the following problems:

1. Determine which pages to compress and when to compress them to minimize performance and energy penalties. This is particularly challenging for low-end embedded systems with tight memory constraints and without MMUs.
2. Control the organization of compressed and uncompressed memory regions and the migration of data between them to maximize the increase in usable memory while minimizing performance and energy consumption penalties.
3. Design a compression algorithm for use in embedded systems that has low performance overhead, low memory requirements, and a good compression ratio for data commonly present in MMU-less embedded systems. For example, data sensed, processed, and communicated in sensor network nodes, such as audio samples, light levels, temperatures, humidities, and, in some cases, two-dimensional images.

MEMMU divides physical RAM into three regions: the reserved region, the compressed region, and the uncompressed region. The reserved region is used to store uncompressed data of the OS, data structures used by MEMMU, and small data elements. The compressed region and the uncompressed region are both used by applications. Application data are automatically migrated between the compressed and the uncompressed regions. The size of each region is decided by compile-time analysis of application memory requirements and estimated compression ratio. The compressed region can be viewed as a high-capacity but somewhat slower form of memory, and the uncompressed region can be viewed as a small, high-performance data cache.

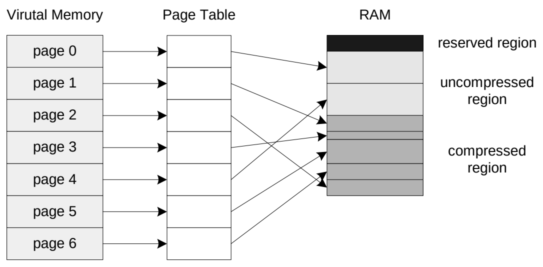


Figure 5.1: Memory layout.

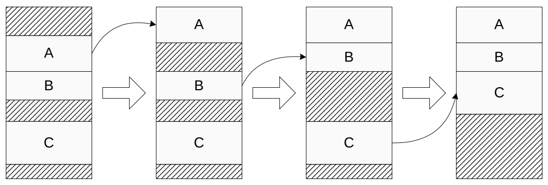


Figure 5.2: Memory coalescing.

Figure 5.1 illustrates the memory layout of an embedded system using MEMMU. From the perspective of application designers, all memory in the left-most *Virtual Memory* column is available. Virtual memory is broken into uniform-sized regions called pages. These pages are mapped to the uncompressed or compressed region (shown to the right of Figure 5.1) via a software-maintained page table. The page number is used as an index into the page table. A memory management mechanism was designed to manage data compression, decompression, and migration between the two regions.

5.4.1 Handle-Based Data Access

Data elements are accessed via their virtual address handles. The virtual page number of a corresponding virtual address is obtained by dividing the virtual address by the page size. The mapping from virtual page to RAM is stored in a page table maintained as an array. For example, if the content of index n in the array is m, and m is in the range of uncompressed pages, virtual page n is mapped to page m in the uncompressed region. If m is greater than number of uncompressed pages, n is mapped to a page in the compressed region.

When data are accessed via their virtual addresses within an application, MEMMU first determines the status of the corresponding virtual page based on the page table.

1. If the virtual page maps to an uncompressed page, the physical address can be di-

rectly obtained by adding the offset to the address of the uncompressed page. The data element is then accessed via its physical address.

2. If the virtual page has not been accessed before, i.e., no mapping has yet been determined for the virtual page, a mapping from this page to an available page in the uncompressed region is created. If there is no available page in the uncompressed region, a victim page is moved to the compressed region to make an uncompressed page available.

3. If the virtual page maps to a compressed page, the page is decompressed and moved to the uncompressed region. Again, if there is no available page in the uncompressed region, a victim page is moved to the compressed region to make space for an uncompressed page available.

In order to make the procedure transparent to users, and to avoid increasing application development complexity, the routines for these operations are stored in a runtime library and compiler transformations are used to convert memory accesses within unmodified code to library calls. Figure 5.3 illustrates the `write_handle` procedure. The three vertical paths prior to the final store instruction correspond to the situations discussed above. The left path shows the case in which a virtual page *p0* maps to a page *PT[p0]* in the uncompressed region. Its physical address is computed by adding offset to the physical page address. In the other two paths, virtual page *p0* maps to a compressed page. More specifically, in the middle path, a free page *p1* is available in the uncompressed region. The compressed page is decompressed to *p1* and a mapping from *p0* to *p1* is created in the page table. Otherwise if the uncompressed region is full, as shown in the right path, a victim page *p2* from the uncompressed region is compressed. In that case, the physical page previously used by *p2* is freed and is now used to store decompressed *p0*. Finally,

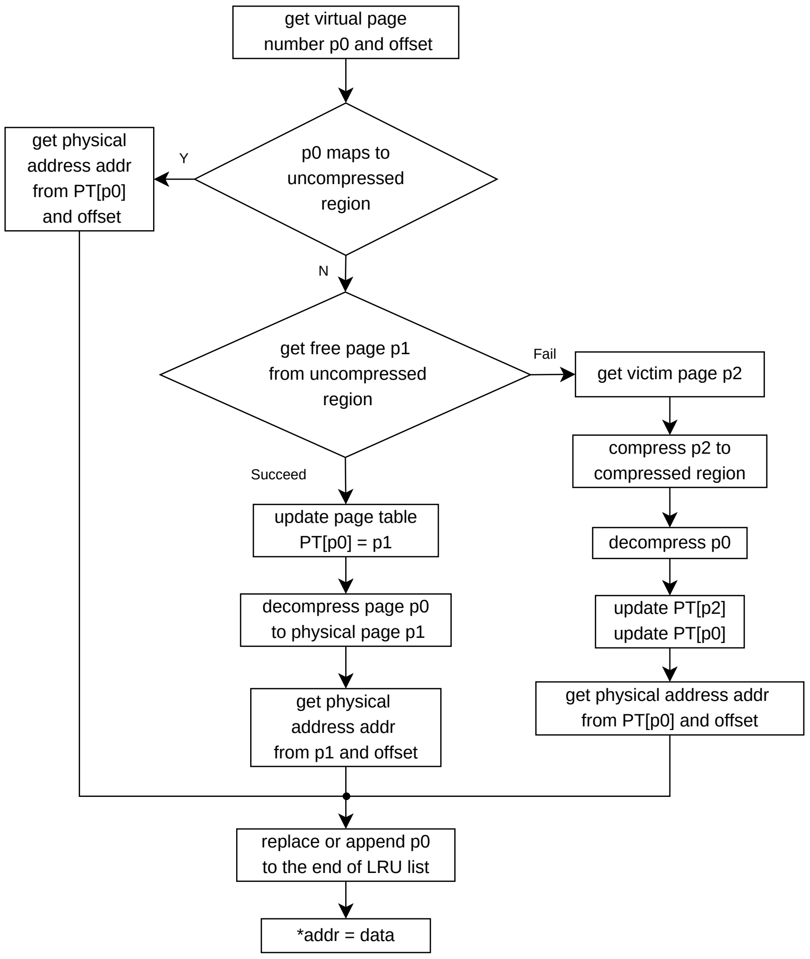

get virtual page number p0 and offset
p0 maps to uncompressed region
Y
get physical address addr from PT[p0] and offset
N
get free page p1 from uncompressed region
Fail
get victim page p2
compress p2 to compressed region
decompress p0
update PT[p2] update PT[p0]
get physical address addr from PT[p0] and offset
Succeed
update page table PT[p0] = p1
decompress page p0 to physical page p1
get physical address addr from p1 and offset
replace or append p0 to the end of LRU list
*addr = data

Figure 5.3: Write handle procedure.

p0 is mapped to a physical page in the uncompressed region and data are written to the physical address.

5.4.2 Memory Management and Page Replacement

When the uncompressed memory region is filled by an application, its pages are incrementally moved to the compressed region to make space available in the uncompressed region. When data in the compressed region are later accessed, they are decompressed and moved back to the uncompressed region. Ideally, pages that are unlikely to be used for a long time should be compressed to minimize the total number of compression and decompression events. MEMMU approximates this behavior via an LRU victim page selection policy. The LRU list is doubly-linked. Every item in the LRU list stores the associated virtual page handle. Handles are ordered by the sequence of handle references. When a page that is already in the LRU list is accessed, it is relocated to the tail of the list, otherwise the new page is appended to the list. The page at the head of the LRU list is selected for compression. After a victim page is compressed, the corresponding node is removed from the LRU list. Therefore, page handles in the LRU list indicate pages currently residing in the uncompressed region.

Managing the uncompressed memory region is straightforward since pages have uniform sizes. On the contrary, managing the compressed region is complex since page sizes differ. Dynamic memory allocation is used in the compressed region to permit the immediate reuse of space when a page is decompressed and moved back to the uncompressed region. Compressed memory management is akin to heap management. It imposes memory overhead for keeping information such as page sizes and addresses (refer to Section 5.5 for MEMMU's memory overhead). This overhead is important in embedded systems that contain only a few kilobytes of RAM. We use the *best fit* policy, which allocates the smallest free slot equal to or larger than the required size. *Best fit* tends to produce the least fragmentation and minimizes the performance overhead resulting from splitting and merging free slots. Pages that are moved from the compressed region to the uncompressed region to read data, and returned to the compressed region without changes have the same compressed size. As a result, they can often be returned to their prior locations in the compressed region, in which they fit exactly. In this case, no free slot merging or splitting will occur. Though *best fit* needs to scan the whole free slot list, the performance overhead is low because the number of free slots, which is upper-bounded by the number of compressed pages, is small.

5.4.3 Preventing Fragmentation

Fragmentation is frequently a problem for dynamic memory allocation techniques. Fragmentation can prevent a newly compressed page from fitting in the compressed region even though the total available memory in that region is sufficient. This situation has the potential to terminate application execution. MEMMU performs *memory merging* and *coalescing* to prevent fragmentation.

Free block merging takes place every time a page is decompressed and removed from the compressed region. Free block handles are maintained in a list in order of the physical address of the compressed areas. If a free block is adjacent to its predecessor or successor, these adjacent blocks are merged. This is a well-known memory management technique.

Coalescing occurs when the memory allocator fails to allocate a new block from the free list. In this case, MEMMU locates pages in order of increasing addresses and moves them to the top of the compressed region, or to the bottom of the most-recently moved pages. This process continues until all compressed pages have been moved. Upon completion, a single large free region remains. Figure 5.2 illustrates this procedure. Rectangles A, B, and C represent three compressed pages and shaded rectangles represent freed blocks. Initially, a request for a size a little bigger than the first free block cannot be satisfied because these free blocks are not continuous. After three iterations of moving A, B, and C upward, all freed blocks are merged into one big free block, and the requested block can be allocated from the big free block. This coalescing algorithm has a time complexity of $\mathcal{O}\left(n^2\right)$, where n is the total number of compressed pages. However, since in practice n is usually small, the cost of coalescing is low. For example, a TelosB mote with 10 KB RAM and a page size of 256 bytes has 40 pages of RAM. In addition to the three pages used for the reserved region (one page used by the operating system and two pages used by MEMMU), it may need 18 compressed pages ($n = 18$) and 19 uncompressed pages to expand the usable memory by $(18/0.5 + 19)/(40 - 1) - 1 = 41\%$. Note that coalescing never imposes a performance penalty unless it is the only remaining alternative permitting the allocation of needed memory. It improves usable memory size for multiple benchmark applications.

5.4.4 Interrupt Management

The primary target platform for MEMMU is wireless sensor network nodes, which are typically memory-constrained, MMU-less embedded systems. On sensor nodes, hardware interrupts often take place when newly-sensed data arrive. There are two naive approaches to handle interrupts during page misses: (1) disable them when accessing data in memory or (2) allow interrupts at any time. Unfortunately, the first approach would result in interrupt misses when interrupts occur during page misses; the second approach is also dangerous because any access to a page in the compressed region during the execution of an interrupt service routine triggered during a page miss would result in an inconsistent compressed region state. In this section, we describe the potential for missed interrupts in more detail and propose a solution.

Consider an environmental data sampling application in which missing samples is not acceptable. Although the optimization techniques described in Section 5.4.5 can be used to reduce the overall execution time overhead, they cannot reduce the worst-case data access delay. In the worst case, the pages of data (except the control data structures stored in the reserved region) referenced in the sampling event handler are all in the compressed region, but there is neither available space in the uncompressed region to decompress these pages nor space in the compressed region to compress a victim page. In this situation, coalescing, compression, and decompression must be performed before each data reference, i.e.,

$$\textit{worst\_case\_delay} = N \times (t_{\textit{coalesce}} + t_{\textit{comp.}} + t_{\textit{decomp.}})$$

where the t values are durations and N is the number of memory references in the sampling event handler. For most applications, the action taken on a sampling interrupt is merely storing the sensed data. Other tasks are posted to process the data later. Therefore, the interrupt handler only has one memory reference that may point to the compressed region. The worst-case coalescing time is encountered when all blocks in the compressed region must be moved upward. This latency can be bounded by the time required to copy the entire compressed region plus the time required by the coalescing algorithm. We measured the worst-case delay on a TelosB wireless sensor node described in Section 5.4.4, assuming the compression algorithm introduced in Section 5.4.6 is used. The time required to compress and decompress one 256 byte page is 3.2 ms. The worst-case coalescing delay on a TelosB mote with a compressed region of 20 pages is 15.7 ms. MEMMU should only be used for applications in which the worst-case delay does not violate any hard timing constraints. If the data set accessed in the interrupt handler is small, this delay can be avoided by storing this data set in the reserved region. This is normally the case because the data set is generally a small buffer.

In applications that compute only in response to sampling events, samples will not be missed if the sampling period is longer than the worst-case compression and decompression delay triggered by a sampling event. However, constraining sampling rate is not always an acceptable solution because some applications may require high sampling rates and even infrequent events may occur during a page miss. To solve this problem, a *ring buffer* may be used. The ring buffer sits in the reserved memory region. When data arrive, they are immediately stored in the ring buffer and a `process_rbuf` task is posted, which moves older data in the ring buffer to the sample buffer. This technique prevents data that arrive during page misses from being dropped. The ring buffer should be large enough to hold the longest-possible sequence of missed samples. Our experiments indicate that an application sampling at 19,600 bps, i.e., 2,450 sample per second, requires a ring buffer of at most 20 bytes. The use of a ring buffer for high-frequency sampling applications is the only portion of the proposed design flow that requires (minor) changes to user application code. Note that MEMMU does not require the use of a ring buffer when sampling rate is low or when missing some samples is acceptable. MEMMU provides ring buffer as a convenient and low-overhead method of preventing missed interrupts when necessary. In order to use ring buffer, one needs to set the ring buffer length based on estimated worst-case delay, insert the `write_rbuf` function call, and post the `process_rbuf` task to transfer data from ring buffer to the application data structure. This approach also solves the problem described at the beginning of Section 5.4.4. By using a ring buffer, the interrupt handler does not access pages in the compressed or the uncompressed regions, so there is no concern for a race condition. This approach also applies to other types of interrupts.

5.4.5 Optimization Techniques

In previous sections, we described the basic design components of the MEMMU memory expansion system. With basic, unoptimized MEMMU, every memory access requires

1. A runtime handle check to determine whether the address being accessed is in the uncompressed region;
2. Compression and decompression if the address is not in the uncompressed region;
3. An update to the LRU list; and
4. Virtual to physical address translation, which includes reading the physical page number from the page table, and operations such as shift and add.

This introduces high execution time overhead that is proportional to the total number of memory accesses. Hence, the basic software virtual memory solution is not practical for many real applications on embedded systems. However, optimization techniques can be used to significantly reduce the number of runtime checks, LRU list updates, and address translations. In this section, we describe several such compile-time optimization techniques. Many of these optimizations are related to existing compiler analysis and loop transformation work [93, 10, 85]. The proposed optimization techniques are based on the analysis of explicit array access. This will pose no problem for most sensor networking applications. For example, almost all of the contributed applications in the TinyOS source repository use explicit array access. These applications were contributed by numerous research and industry teams. If applications include implicit array accesses via pointers, existing compiler techniques could be used to transform them to explicit accesses [35,140]. This compiler transformation is not currently supported by LLVM. However, it would be trivial to use such a compiler pass in MEMMU as soon it becomes available.

1. **Small object optimization:** If a small data element is used very frequently in the application, it should be assigned to the reserved region at compile time to eliminate all related handle checks and address translations. The increase in usable memory resulting from allowing the migration of small globals, such as scalars, is generally not sufficient to offset the cost of managing their migration. For example, in the image convolution application shown in Figure 5.4(a), the small matrix of coefficients, K, is accessed in every iteration of the loop (line 8) and the size of this matrix is small. After moving it to the reserved region, we can eliminate $(W - M + 1) \times (H - M + 1) \times M \times M$ runtime checks and address translations related to this matrix. Using a reserved region also prevent infrequently used data from occupying the uncompressed region because they are stored in the same page

with frequently referenced data. The *small object optimization* is implemented by modifying LLVM [64] to allocate all data structures smaller than a threshold in the reserved region since their sizes add up only to a few percent of the memory required by the application.

2. **Runtime handle check optimization:** This technique is based on the observation that if a sequence of memory references access the same page, only the first handle check is necessary since the referenced page is sure to be in the uncompressed region on subsequent accesses. This optimization is specific to sequential access patterns, although different increment and decrement offsets are supported. By inserting checks to decide whether the data element to be accessed next is in a different page from the previous one, the number of handle checks for all accesses to the same page can be reduced to one. Performance is improved because the inserted check is relatively faster than reading an element from the array (page table). This can be especially useful for a hardware-triggered sample arrival event that writes data into the buffer, as illustrated by Figure 5.6. *Data_ready* is a hardware-triggered event. The *if* statement in the optimized code in Figure 5.6(b) filters all the handle checks mapping to the same page that was checked in the previous reference.

   The *Runtime handle check optimization* takes place in a compiler pass, in which LLVM creates two global variables, current page number and previous page number, for each `check_handle` and puts every `check_handle` call in an *if* statement. `Check_handle` is called only when the current page number differs from the previous page number. This technique may introduce overhead in some applications, such as an application that accesses interleaved pages, because the current page number will always be different from the previous page number. Therefore, it is only applied to programs or sections of code that access one array with affine function of induction variables. Affine functions represent vector-valued functions of the form $f(x_1, ..., x_n) = A_1x_1 + ... + A_nx_n + b$.

3. **Loop transformation and compile-time elimination of inner-loop checks:** This optimization scheme further reduces runtime handle checks via compile-time loop transformations. It may be applied to loops whose array accesses are affine functions of enclosing loop induction variables. Figure 5.5(a) illustrates an example of sequential references to an array. At most *PAGESIZE* references access the same page. Figure 5.5(b) illustrates the unoptimized solution, which inserts a handle check before every memory reference (line 2) and replaces writes to memory with calls to the `write_handle` routine (line 3). The entire loop requires N handle checks. Figure 5.5(c) illustrates an optimized solution. Loop transformation is used to break the original loop into nested loops. Iterations of the inner loop (line 4) access memory inside a single page. Therefore, handle checks for the inner loop can be replaced by one check in the outer loop (line 3). The total number of handle checks is reduced from N to $N/$*PAGESIZE*. For the sake of simplicity, array A shown in Figure 5.5 is page-aligned. This loop transformation is a type of loop tiling [93].

   The loop transformation technique can also be applied in the following, more general, circumstances.

   (a) The loop accesses only one array and the offset is a linear function of the loop induction variable. In the transformed code, every exit from the inner loop implies that the next accessed address is in a different page. When *PAGESIZE* is evenly divided by the stride, the number of iterations of inner loop is constant: *PAGESIZE* divided by stride. However, the number of inner loop iterations varies if the *PAGESIZE* is not evenly divided by the stride. In that case, variables *start* and *end* are used to control the iteration count for the inner loop by locating the offset in the referenced page at the beginning or end of the inner loop. Example code is shown in Figure 5.7. *Start* is calculated via modular division of the first address by *PAGESIZE*; *end* is obtained via modular division of the largest address by *PAGESIZE* for the last iteration and by *PAGESIZE* for other iterations.

(b) The loop accesses n arrays with the same stride and $2 \times n - 1$ is no larger than the number of pages in the uncompressed region m. Figure 5.10 shows how a loop accessing arrays A, B, and C is transformed. The numbers in the arrays correspond to virtual page indices. The original loop carries out interleaved accesses to these arrays, from the top to the bottom. The loop is divided based on the page boundaries in the array in which a page boundary is first crossed. The arrows beside array C indicate iterations of the transformed loop. The numbers to the right of the arrows are the pages brought into the uncompressed region before each iteration. For example, at the beginning of third iteration, pages 2, 8, and 14 are brought into the uncompressed region. Pages 7 and 13 should not be compressed, because they will be accessed during the second iteration. The dashed box in Figure 5.10 indicates all of the pages accessed during one iteration. Clearly, regardless of the vertical position of the box, it can overlap at most $2 \times (n - 1) + 1$ pages. Therefore, this is the maximum number of pages required in the uncompressed region.

(c) If the loop accesses multiple arrays with different strides, only perform transformation on the arrays that meet conditions 1 or 2, above.

4. **Handle check hoisting:** Hoisting handle checks is the process of replacing multiple handle checks inside a loop with one handle check outside the loop. This optimization requires that the total size of the accessed pages is no larger than the size of the uncompressed region. It can be viewed as prefetching pages before entering the loop and locking them in the uncompressed region until an iteration of the loop finishes execution. The smallest and largest addresses accessed for each memory object during one iteration are obtained and the largest possible number of pages between them is computed. Figure 5.4 gives an example of handle check hoisting. Figure 5.4(a) is the original code for image convolution. Without handle check hoisting, MEMMU requires $(H - M + 1) \times (W - M + 1) \times (2 \times M \times M + 1)$ handle checks. It can be decided at compile time that the second inner loop (line 3), which covers three rows of A and one row of B, is the largest loop that can reside in the uncompressed region. Therefore, handle checks are hoisted to the beginning of the second inner loop, as shown in Figure 5.4(b) line 3. This eliminates at least $(H - M + 1) \times (W - M + 1) \times (2 \times M \times M + 1) - (H - M + 1) \times 4$ handle checks. Note that at most four pages may be covered in the second loop, two for each array. To maximize performance while maintaining correctness, we start from the inner-most loop, and expand outward until the analyzed memory usage in the next loop cannot be accommodated in the uncompressed region or we reach the outer-most loop.

5. **Pointer dereferencing to reduce address translation:** The purpose of the *pointer dereferencing* optimization is related to that of strength reduction optimizations [93]: replacing expensive operations with less expensive operations. In particular, it replaces calls to `write_handle` and `read_handle` functions that contain complicated operations for address translation to pointer dereferencing with simple pointer

computations. Assume the accessed virtual address is an affine function of a basic induction variable i: $a \times i + b$, a and b are constants. The physical address of the memory reference in question is $phy\_addr = PT[(A + a \times i + b)/PAGESIZE] + (A + a \times i + b)\%PAGESIZE$. $PT[(A + a \times i + b)$ computes the starting address of the physical page, $(A + a \times i + b)\%PAGESIZE$ computes the offset in the page. Normally, this operation cannot be optimized by general strength reduction optimizations. However, if we know that the succeeding reference is in the same page and the state of the page does not change between the references, this operation can be reduced to $phy\_addr+ = a \times i.diff$, where $i.diff$ is the constant change for i during each iteration of the loop. Therefore, *pointer dereferencing* is used after *runtime handle check optimization* or *loop transformation*. During *runtime handle check optimization*, each time a new page is accessed, i.e., inside the *if* statement, a base pointer is computed; the following accesses in the same page dereference the base pointer instead of referring to the page table. After *loop transformation*, before entering the inner loop, base pointers are computed, and addresses accessed in the inner loop are computed by dereferencing the base pointer. Figure 5.5(d) shows that this optimization scheme, which is implemented in line 4, 6, and 7, can eliminate $N - N/PAGESIZE$ address translations. The *pointer dereferencing* optimization replaces calls to the `write_handle` and the `read_handle` functions with direct access via a pointer.

Each application may have a different set of effective optimizations, as shown in Section 5.5.7. The following policy is followed by MEMMU to determine the optimizations to use for a given application:

1. Apply *small object optimization* during the instruction replacement pass by leaving reads and writes of small data structures unchanged.

```
Input: 2-D array $A[H, W]$
Input: 2-D array $K[M, M]$
Output: 2-D array $B[H - M + 1, W - M + 1]$
 1: $n \leftarrow \sum_{p=0}^{M-1} \sum_{q=0}^{M-1} K_{pq}$
 2: for $i \in \{0 \cdots H - M\}$ do
 3:     for $j \in \{0 \cdots W - M\}$ do
 4:         $t \leftarrow 0$
 5:         for $a \in \{0 \cdots M - 1\}$ do
 6:             for $b \in \{0 \cdots M - 1\}$ do
 7:                 $p \leftarrow A[i + a][j + b]$
 8:                 $q \leftarrow K[a][b]$
 9:                 $t \leftarrow t + p \times q$
10:             end for
11:         end for
12:         $B[i][j] \leftarrow t/n$
13:     end for
14: end for
```

(a)

```
Input: array $A$ allocated by vm_malloc($H \times W$)
Input: 2-D array $K[M, M]$
Output: array $B$ allocated by
    vm_malloc($(H - M + 1) \times (W - M + 1)$)
 1: $n \leftarrow \sum_{p=0}^{M-1} \sum_{q=0}^{M-1} K_{pq}$
 2: for $i \in \{0 \cdots H - M\}$ do
 3:     Bring in pages to be used in the following loop
        to uncompressed region
 4:     for $j \in \{0 \cdots W - M\}$ do
 5:         $t \leftarrow 0$
 6:         for $a \in \{0 \cdots M - 1\}$ do
 7:             for $b \in \{0 \cdots M - 1\}$ do
 8:                 $p \leftarrow$ read_handle($A + (i+a) \times W + j + b$)
 9:                 $q \leftarrow K[a][b]$
10:                 $t \leftarrow t + p \times q$
11:             end for
12:         end for
13:         write_handle($B + i \times (W - K + 1) + j,\ t/n$)
14:     end for
15: end for
```

(b)

Figure 5.4: Example of (a) original and (b) transformed convolution application.

2. Apply *loop transformation* to a loop if the referencing array index is a linear function of the induction variable. Then apply *pointer dereferencing*.

3. If the second step is not used for the application, then try *handle check hoisting*.

4. If neither the second nor third steps are used, and the loop only accesses a single array sequentially, apply the *runtime handle check optimization* and *pointer dereferencing*.

The above policy implies a priority order on the proposed optimization techniques. However, this selection order is a heuristic and may not be optimal. Each step is provided in a separate compiler pass. Therefore, one might potentially run the passes in another order to find out the optimal solution for a particular application.

5.4.6 Delta Compression Algorithm

We developed a high-performance, lossless compression algorithm based on delta compression for use in sensor network applications. This algorithm exploits the simi-

```
Variable: array $A[N]$
for $i \in \{0 \cdots N\}$ do
    $A[i] \leftarrow x$
end for
```

(a) Original code.

```
Variable: array $A$ allocated by vm_malloc($N$)
for $i \in \{0 \cdots N\}$ do
    check_handle($(A + i)/PAGESIZE$)
    write_handle($A + i,\ x$)
end for
```

(b) Transformed code without optimization.

```
Variable: array $A$ allocated by vm_malloc($N$)
$pnum \leftarrow A/PAGESIZE$
for $i \in \{A/PAGESIZE \cdots (A + N)/PAGESIZE\}$ do
    check_handle($pnum$)
    for $j \in \{0 \cdots PAGESIZE\}$ do
        write_handle($A + i \times PAGESIZE + j,\ x$)
    end for
    $pnum++$
end for
```

(c) Transformed code with loop transformation.

```
Variable: array $A$ allocated by vm_malloc($N$)
$pnum \leftarrow A/PAGESIZE$
for $i \in \{A/PAGESIZE \cdots (A + N)/PAGESIZE\}$ do
    check_handle($pnum$)
    $base\_ptr \leftarrow$ virtual_to_physical($A + i \times PAGESIZE$)
    for $j \in \{0 \cdots PAGESIZE\}$ do
        $*base\_ptr \leftarrow x$
        $base\_ptr++$
    end for
    $pnum++$
end for
```

(d) Transformed code with loop transformation and pointer dereferencing.

Figure 5.5: Example of optimizations on an array accesses.

```
check_handle($buf + count$)
write_handle($buf + count,\ data$)
$count++$
```

(a) Original code.

```
$cur\_page \leftarrow (buf + count)/PAGESIZE$
if $cur\_page \neq last\_page$ then
    check_handle($cur\_page$)
end if
write_handle($buf + count,\ data$)
$count++$
$last\_page \leftarrow cur\_page$
```

(b) Transformed code with runtime handle check optimization.

Figure 5.6: Example code transformation of data_ready() function.

```
Variable: array $A[M]$
for $i \in \{0 \cdots N\}$ do
    $A[i \times a + b] \leftarrow x$
end for
```

(a) Original code.

```
Variable: array $A$ allocated by vm_alloc($M$)
for $i \in \{0 \cdots N\}$ do
    $page \leftarrow (A + i \times a + b)/PAGESIZE$
    check_handle($page$)
    write_handle($A + i \times a + b,\ x$)
end for
```

(b) Transformed code without optimization.

```
Variable: array $A$ allocated by vm_alloc($M$)
$t \leftarrow A + b$
$p\_min \leftarrow (A + b)/PAGESIZE$
$p\_max \leftarrow (A + a \times N + b)/PAGESIZE$
for $page \in \{p\_min \cdots p\_max\}$ do
    check_handle($page$)
    for $j \in \{start \cdots end\}$ do
        write_handle($t,\ x$)
        $t \leftarrow t + a$
        $j \leftarrow j + a$
    end for
end for
```

(c) Transformed code with loop transformation.

Figure 5.7: Loop transformation on sequential memory access with constant stride.

```
Input: IN word stream
Output: OUT word stream
Variable: DATA word stream, TAPE delta stream
1: for $i \in \{1, \cdots, N\}$ do
2:     $\delta \leftarrow$ IN[i] - IN[i-1]
3:     if $\log_2 \delta \leq$ MAXBITS then
4:         TAPE[i] $\leftarrow \delta$
5:     else
6:         TAPE[i] $\leftarrow$ MAGIC_CODE
7:         DATA[i] $\leftarrow$ IN[i]
8:     end if
9:     OUT $\leftarrow$ pack(TAPE, DATA)
10: end for
```

```
Input: IN word stream
Output: OUT word stream
Variable: DATA word stream, TAPE delta stream
1: DATA, TAPE $\leftarrow$ unpack(IN)
2: for TAPE[i] in range of TAPE do
3:     if TAPE[i] = MAGIC_CODE then
4:         OUT[i] $\leftarrow$ DATA[i]
5:     else
6:         $\delta \leftarrow$ TAPE[i]
7:         OUT[i] $\leftarrow$ OUT[i-1] + $\delta$
8:     end if
9: end for
```

Figure 5.8: Delta compression and decompression.

larities between adjacent data elements. Despite its simplicity, the algorithm has high performance and a good compression ratio for sensor data in which adjacent samples are often correlated.

To design an appropriate compression algorithm for sensor data, the regularities of the data must be well understood. For this purpose, we collected numerous types of sensor data, e.g., sound, light, and temperature, from Crossbow MICAz and TelosB sensor network nodes and analyzed their characteristics. Intuitively, sensor data are likely to stay similar during a certain period of time, and within a certain geographic range, hence showing high amounts of temporal and spatial locality. For example, in sensor networks deployed for seabird habitat monitoring [108] sensor nodes may be placed in petrel nests in underground burrows. The temperature and humidity sensed from one sensor node usually changes smoothly during a day, except as a result of storms. In addition, the sensor data of temperature and humidity from adjacent burrows are likely to be similar; these data are usually transmitted within a cluster of nodes before they are sent to the base station. Thus, sensor nodes commonly receive highly-redundant data.

A delta-based compression algorithm exploits regularity in data: the difference between two adjacent data elements (delta) usually requires fewer bits to store than the original data [33]. Our implementation of the delta compression and decompression algorithms are presented in Figure 5.8. The algorithms are based on the observation that the majority of the deltas can be stored within a pre-defined *MAXBITS*; if the delta cannot be stored within *MAXBITS*, i.e., there is a sudden change in sensed data, the raw data are stored and a *MAGIC_CODE* is recorded to indicate this abnormality. The algorithm also adapts to the compressibility of pages by means of early termination. When the number of deltas that exceed *MAXBITS* is above a certain threshold, causing the "compressed" page to exceed its original size, the algorithm terminates and reports the compressed page size as zero, indicating that this page is not compressed.

In order to identify the *MAXBITS* value that provides the best compression ratio, we analyzed the sample sound data collected by the Crossbow MICAz sensor node. Since the analog-to-digital converter (ADC) on the MICAz generates a 10-bit output, the compression algorithm reads in 2 bytes (16 bits) at a time and computes the delta on a 2 bytes basis. Figure 5.9 shows that 95% of the deltas can be represented using six bits. Therefore, in our implementation, *MAXBITS* is set to six. Please note that this value may vary depending on the underlying hardware of the sensor node, i.e., the bit width of the ADC.

5.4.7 Page State Preservation

The optimization techniques proposed in Section 5.4.5 improve performance by eliminating runtime handle checks and address translations associated with memory references to pages that have been brought into the uncompressed region. They depend on compile-time knowledge and assignment of page status. However, in an event-driven system where an interrupt can preempt a task, an interrupt handler can potentially cause the compression of a page that is being used by a task. If the task resumes after the location of the page changes, an error would occur. This makes the *loop transformation* and *handle check hoisting* optimizations unusable. To resolve this problem, we lock

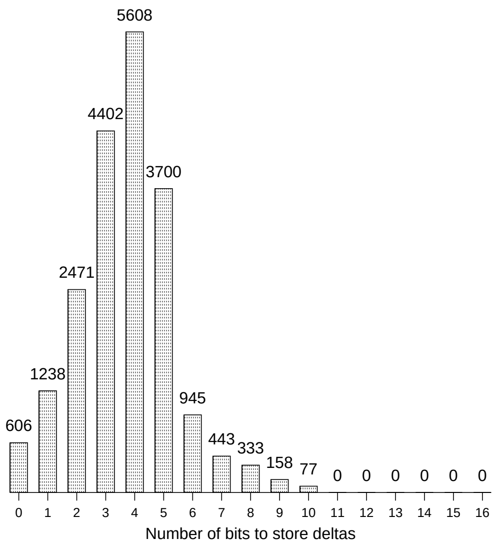


Figure 5.9: Histogram of sensor data delta values.

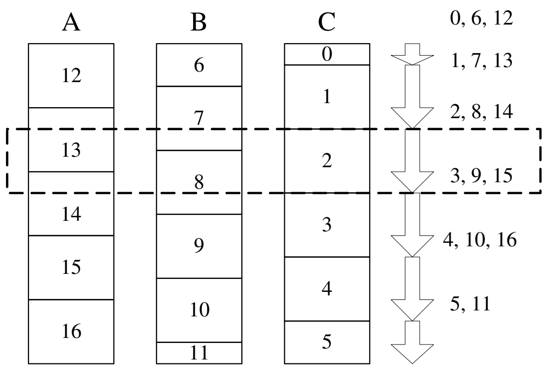


Figure 5.10: Example of loop transform on multiple arrays.

pages for which memory references are optimized in the uncompressed region. This is done by introducing a one-bit flag for each page in the LRU list to indicate whether it is locked. Procedures `lock_handle` and `unlock_handle` are added to MEMMU library to lock a page in the uncompressed region and release the lock. When interrupt handlers can access memory objects outside the reserved region, *loop transformation* and *handle check hoisting* need to replace `check_handle` with `lock_handle` and insert `unlock_handle` after exiting from the optimized inner loop. For example, in Figure 5.5(c) and (d), `check_handle(pnum)` in line 3 will be replaced with `lock_handle(pnum)` and `unlock_handle(pnum)` will be inserted after line 6 and line 8 respectively. In TinyOS, tasks do not preempt each other, so the page locking strategy is only required when interrupts can cause data to be moved between the memory regions. In other words, if after applying *small object optimization* and the ring buffer technique, interrupt handlers only access memory objects in the reserved region, all the optimization techniques discussed in Section 5.4.5 will still be effective. The page state

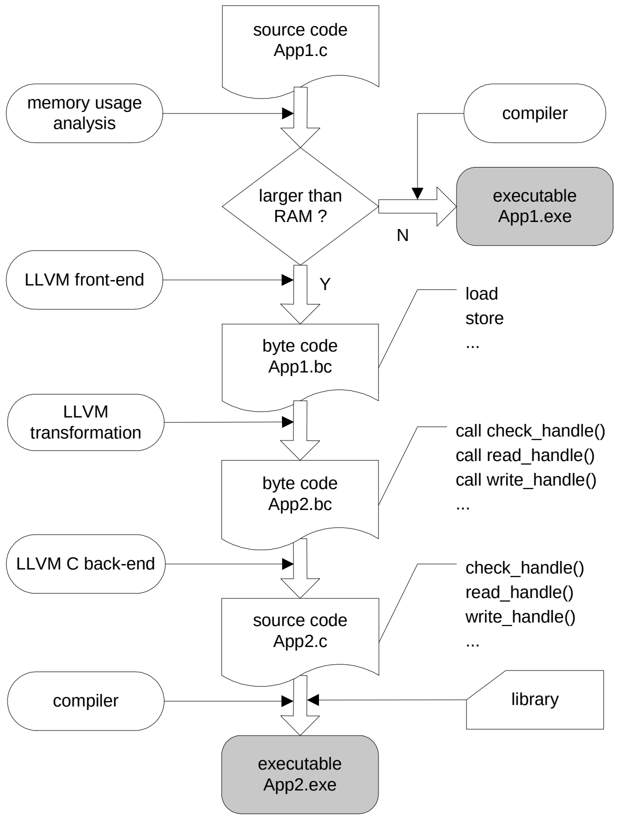


Figure 5.11: Overview of technique.

preservation strategy can be generalized to multi-threaded system by locking pages currently used by each thread. However, the concurrent execution of many threads accessing different pages may degrade the memory expansion ratio by requiring a larger uncompressed region to allow pages used by threads simultaneously to stay uncompressed.

5.4.8 Summary

Figure 5.11 illustrates the procedure for using the MEMMU system to automatically generate an executable from mid-level or high-level language source code such as ANSI C. First, the memory requirements of the application are analyzed. If these requirements are smaller than physical RAM, compression is not necessary and therefore no transformations are performed. Otherwise the application code is compiled to LLVM byte code by the LLVM compiler. After that, memory load and store instructions are replaced with calls to our handle access functions, i.e., `check_handle`, `read_handle`, and `write_handle`. Other transformations are performed to enable the optimizations described in Section 5.4.5. A call to a memory initialization routine is also inserted at the beginning of the byte code. The modified byte code is then converted back to high-level language via the LLVM back-end. Finally, the modified application is compiled with the extended library containing our handle access functions to generate an executable.

In the memory initialization routine, physical memory is divided into three regions. The size of each region is computed based on the application memory requirement and the estimated compression ratio of MEMMU, i.e., the average compression ratio for the many pages of data that may be in use at any point in time. Since the runtime data compression ratio cannot be accurately decided at compile time, it is possible for the runtime compression ratio to be worse than the predicted compression ratio, causing execution to stop when both memory regions are full. Therefore, it is suggested that users determine the compression ratio based on sample data of their application and set the MEMMU compression ratio appropriately. This process could potentially be automated by running the selected compression algorithm on sample data sets. Knowing the exact memory requirement of the original program and the data compression ratio at compile time allows MEMMU to determine the sizes of the compressed and the uncompressed regions to ensure sufficient usable memory for the modified program with minimal performance overhead. Otherwise if this information is not available at compile time, an overestimate in required memory size may result in larger performance overhead and an underestimate in required memory size may result in runtime out-of-memory failure. In Section 5.5.8, we demonstrated that it is easy to compute a tight upper bound on the aggregated compression ratio using training data.

For any compression algorithm, it is possible to construct an input that will result in a compression ratio greater than one. Similarly, given any predicted application average compression ratio, it is possible to construct a sequence of inputs on which compression will exceed the ratio. The frequency of encountering such a sequence of inputs in the field depends strongly on the application. For many applications, such an event will be rare. For example, the compression ratio for individual pages of the vibration data and temperature data shown in Section 5.5.8 never exceed 78.1% and 44.5%, respectively, during 6 months of measurement. Section 5.5.8 also shows that when the estimated compression ratio is set to 1.05$\times$ the average page compression ratio, this results in a very low probability of memory exhaustion for this application: 0.38% or 5.5×10^{-7}% every 30 minutes. Although it is important that the probability of memory exhaustion be low, we believe that it need not be zero in many applications. For example, if this probability is orders of magnitude lower than that of node hardware failure [134], its impact on system reliability will be negligible. If an application required zero probability of memory exhaustion, but the designers still want the functionality and ease-of-design benefits MEMMU can bring, it would be possible to migrate data to secondary storage in the rare event of memory exhaustion, e.g., by using the technique proposed by Choudhuri and Givargis [25]. Combined with MEMMU, this would eliminate the risk of memory overuse at the cost of extremely rare performance penalties when secondary storage must be used.

In our experiments, MEMMU is tested on TelosB motes running TinyOS [40]. TinyOS and its applications are written in nesC [39]. NesC is an extension to the C programming language that supports the structure and execution model of TinyOS. Ncc is the NesC compiler for TinyOS. TinyOS itself does not support dynamic memory allocation, so there are only stack and global variables in the nesC program; this simplifies analysis of application memory requirements.

LLVM does not have a nesC front-end. As a result, one of three possible flows may be used. In the first, a mote development environment based on ANSI C, such as MANTIS OS, may be directly used with LLVM. In the second, the ANSI C computation-intensive portion of the application is manually extracted from the nesC code, provided to LLVM for transformation, and reinserted in the nesC code before compilation with ncc. We used this approach for the experiments presented in Section 5.5. However, we have subsequently developed a fully-automated flow. First, the nesC program is transformed to C by ncc. Then the C program is transformed to byte code by llvm-gcc and MEMMU compiler passes are applied. Finally, the LLVM C-backend transforms the byte code back to a C program and the C program is compiled to an executable by ncc. This flow is complicated by the fact that ncc inserts inline assembly, which LLVM C-backend does not yet support. We have therefore developed a script to temporarily associate inline assembly with dummy function calls, permitting restoration after LLVM transformation passes.

5.5 Experimental Results

This section presents the results of evaluating MEMMU using five representative wireless sensor network applications. These benchmarks were executed on a TelosB wireless sensor node. The TelosB is an MMU-less, low-power, wireless module with integrated sensors, radio, antenna, and an 8 MHz Texas Instruments MSP430 microcontroller. The TelosB has 10 KB RAM and typically runs TinyOS. The benchmarks are tested with three system settings: running the original applications without MEMMU, with an unoptimized version of MEMMU, and with an optimized version of MEMMU. Four metrics were evaluated: average power consumption, execution time, processing rate, and memory usage. We measured total memory usage, memory used by MEMMU, and division between memory regions. Processing rate is defined as application data size divided by execution time.

Table 5.1: Filtering Benchmark

	RAM usage (B)	Buffer size (B)	MEMMU usage (B)	Comp. region (B)	Uncomp. region (B)	Proc. time (s)	Active power (mW)	Average power (mW)
Orig.	9,935	9,728	0	0	0	1.24	6.77	3.94
Unopt.	7,243	9,728	518	3,840	2,560	2.31	6.97	5.92
Opt.	7,243	9,728	518	3,840	2,560	1.35	6.80	4.27

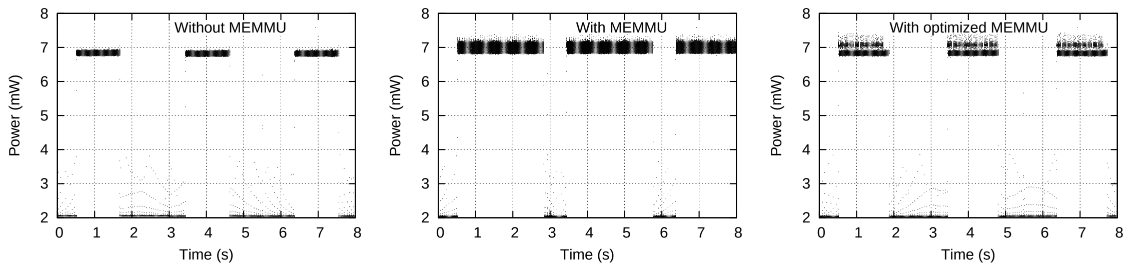


Figure 5.12: Power consumption of the sound-filtering benchmark using three settings.

Power measurements were taken using a National Instruments 6034E data acquisition card attached to the PCI bus of a host workstation running Linux. Power was computed based on the measured voltage across a $10\,\Omega$ resistor in series with the power supply. The average power of duty cycle-based applications is calculated using the following equation.

$$P_{average} = \frac{P_{active} \times t_{active} + P_{idle} \times t_{idle}}{t_{active} + t_{idle}} \tag{5.1}$$

All of LLVM's optimizations are turned off to ensure all the overheads and savings are entirely due to MEMMU. The experimental results show that, with the exception of the image convolution benchmark, the execution time overheads of all other benchmarks are below 10%. Below we will describe each benchmark and discuss the corresponding results in detail.

5.5.1 Sound Filtering

The first example application is sound filtering. When the hardware timer periodically fires, the mote starts one-dimensional filtering on collected audio data. The MSP430 microcontroller automatically puts itself into a low power mode when the task stack is empty and wakes up when the next timer event arrives. As shown in Figure 5.12, the power waveform is similar to a square wave. For this benchmark, we assume fixed application and input data sizes (buffer sizes) and compare the memory usage to determine the amount of memory saved by using MEMMU.

Table 5.1 shows results for this benchmark when running under three system settings. The memory reduction achieved by MEMMU is $9,935 - 7,243 = 2,692$ bytes, which is 27% of the original memory requirement. The saved memory is available to store other data, which may be larger than 2,692 bytes as a result of compression. For this benchmark, *small object optimization*, *loop transformation*, and *pointer dereferencing* were applied. The processing time and active power consumption overheads of unoptimized MEMMU are 86.3% and 3.0%, while after optimization, the overheads are reduced to 8.9% and 0.4%, respectively. Figure 5.12 depicts the power consumption under the three system settings. According to Equation 5.1, there are two causes of increased average power consumption. First, the mote stays in active mode longer when MEMMU is used. Second, active power consumption increases slightly as a result of MEMMU's computations.

Table 5.2 shows the performance overhead from calling MEMMU functions when the optimized version of MEMMU is used. This breakdown in performance overhead was determined by sampling the program counter at a period of 100 Hz during application execution using these data to compute the percentage of execution time spent in each function. Over half of the overhead comes from `compress`. 17.32% and 15.44% may be attributed to `swap_in` and `swap_out`, which contain the instructions to search for

Table 5.2: Overhead of MEMMU Functions

Function name	Compress	Decompress	Swap_in	Swap_out	Check_handle
Overhead (%)	67.07	0	17.32	15.44	0.17

Table 5.3: Convolution Benchmark

	RAM usage (B)	Input image (B)	Output image (B)	MEMMU usage (B)	Comp. region (B)	Uncomp. region (B)	Proc. time (s)	Proc. rate (B/s)	Active power (mW)
Orig.	9,739	4,900	4,624	0	0	0	1.50	6,349	6.57
Unopt.	9,739	6,084	5,776	638	6400	2304	4.47	2,653	6.82
Opt.	9,739	6,084	5,776	638	6400	2304	2.88	4,118	6.75

free pages and update the page list. `Check_handle` calls `swap_in` and `swap_out` if the checked page is compressed and no free page in the uncompressed region is available. `Swap_in` calls `swap_out` if there is no space in the uncompressed region. `Swap_out` calls `compress` to compress a victim page. Note that decompression is very efficient. Therefore the overhead from decompression is close to 0.

We also use this benchmark to evaluate the changes in performance as the memory required by the application increases, i.e., as the memory expansion ratio of MEMMU increases. Figure 5.13 shows the increase in performance (processing rate) as a function of data size in the filtering benchmark using the optimized version of MEMMU. The total physical memory usage stays constant. The left-most point shows the base case, in which the physical memory is sufficient to run the application. In this case, MEMMU is not used. Each of the other points in the figure corresponds to an optimal memory division that minimizes the performance overhead while meeting the memory requirement. The results show that the performance penalty stays almost constant despite increasing application data size. Therefore, even though a larger compression region is needed as application data sets grow, the performance overhead of MEMMU is fairly stable.

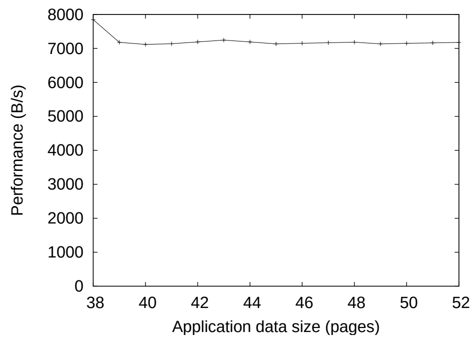


Figure 5.13: Relation between performance and application data size.

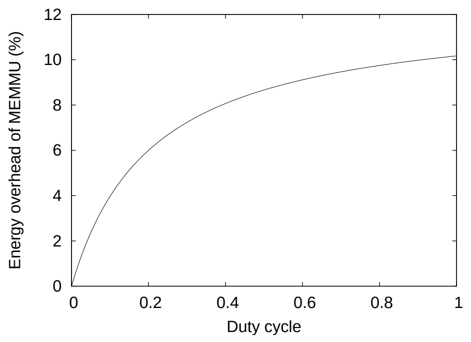


Figure 5.14: Energy overhead of MEMMU as a function of duty cycle.

5.5.2 Image Convolution

Our second example application is a convolution algorithm in which a large matrix is convolved with a 3×3 coefficient kernel matrix. Note that 2-D convolution is used for graphical images. In order to permit consistent input to allow fair comparisons for each test case, the input images were generated by scaling the same image to different sizes; a gray-scale image of a cloudy sky was used. The input images were transferred to the mote via USB. Table 5.3 compares the input and output image sizes, RAM usage, processing rate, execution time, and average power consumption of the benchmark application under three settings. The results indicate that using the same amount of physical RAM, MEMMU allows the application to handle images that require more memory than is physically available: the unmodified TelosB can only handle an input image smaller than 4.8 KB, while MEMMU allows the mote to process images that are 25% larger (6 KB). Since the delta compression algorithm is less efficient for 8-bit images, the compression ratio in this case is 62.4%. We believe a lossy compression algorithm designed for image data would permit a higher usable memory improvement ratio.

Unfortunately, the increase in image size imposes a cost. Using MEMMU results

Table 5.4: Light Sampling Benchmark

	RAM usage (B)	Buffer size (B)	MEMMU usage (B)	Comp. region (B)	Uncomp. region (B)	Proc. time (s)	Proc. rate (B/s)	Active power (mW)
Orig.	9,474	9,040	0	0	0	4.39	2,059	57.44
Unopt.	9,474	13,200	603	5,120	3,328	6.53	2,021	58.61
Opt.	9,474	13,200	603	5,120	3,328	6.47	2,040	58.11

in a 58.2% decrease in processing rate and 3.8% increase in power consumption. After applying *small object optimization* and *handle check hoisting*, the processing rate penalty was reduced to 35.1% and the power consumption penalty was reduced to 2.1%. Please note that the image convolution benchmark was the only benchmark for which MEMMU had a performance overhead higher than 10% after optimization. The performance penalty reduction is smaller compared to other applications because *pointer dereferencing* cannot be used to reduce the penalty caused by address translation.

5.5.3 Data Sampling

The third example application is sensor data sampling. In this application, the mote senses the light level every 1 ms and stores the data to a buffer. When the buffer is full, its contents are sent via the wireless transmitter. *Small object optimization*, *handle check hoisting*, and *pointer dereferencing* were applied to this benchmark. Table 5.4 shows that with MEMMU, the buffer size is increased by 46.0% without increasing physical memory usage. The average power consumption overheads are 2.0% and 1.1% for unoptimized and optimized MEMMU respectively. The processing time and processing rate measure the time and speed of transmitting the data in the buffer. The processing rate is reduced by 1.8% with unoptimized MEMMU. Optimizations reduced the performance overhead to 0.9%.

Table 5.5: Covariance Matrix Computation Benchmark

	RAM usage (B)	Buffer size (B)	MEMMU usage (B)	Comp. region (B)	Uncomp. region (B)	Proc. time (s)	Proc. rate (B/s)	Active power (mW)
Orig.	9,643	9,430	0	0	0	0.47	19,895	5.22
Unopt.	9,643	13,056	602	5,120	3,584	1.44	9,067	5.40
Opt.	9,643	13,056	602	5,120	3,584	0.72	18,133	5.36

Table 5.6: Correlation Computation Benchmark

	RAM usage (B)	Signal size (B)	MEMMU usage (B)	Comp. region (B)	Uncomp. region (B)	Proc. time (s)	Proc. rate (B/s)	Active power (mW)
Orig.	6,669	6,460	0	0	0	7.98	810	5.34
Unopt.	6,669	9,728	543	4532	1536	28.3	344	5.36
Opt.	6,669	9,728	543	4532	1536	13.00	748	5.35

5.5.4 Covariance Matrix Computation

The fourth example application is covariance matrix computation. This application is useful in statistical analysis and data reduction. For example, it is the first stage of principal component analysis. Each vector contains a number of scalars with different attributes, e.g., different types of sensor data. *Small object optimization*, *runtime handle check optimization*, and *pointer dereferencing* were applied to this benchmark. Table 5.5 shows that MEMMU permits more vectors to be processed at a single time: the buffer size increases by 38.5%. Although the performance penalty of unoptimized MEMMU is large (the processing rate is decreased by 54.4%), optimizations reduce it greatly. The processing rate using the optimized version of MEMMU is only 8.9% lower than the original application. The average power consumption penalties of both unoptimized and optimized MEMMU are below 4%.

Table 5.7: Comparison of Optimization Techniques

Benchmark	Run time of benchmarks with different MEMMU optimizations (s)					
	Unopt. MEMMU	Runtime handle check	Handle check hoisting	Loop trans.	Runtime handle check & pointer deref.	Loop trans. & pointer deref.
Filtering	1.84	1.25	1.30	1.18	1.20	1.12
Sampling	5.39	5.38	N.A.	N.A.	5.37	N.A.
Correlation	21.11	22.50	N.A.	22.53	15.20	12.94
Covariance	1.12	0.86	0.83	N.A.	0.53	N.A.
Convolution	2.88	2.63	1.97	N.A.	N.A.	N.A.

Table 5.8: Code Size Overhead Introduced by MEMMU

Code size	Filtering	Convolution	Sampling	Covariance	Correlation
Original (B)	16020	16725	15282	16400	16919
With MEMMU (B)	20888	21882	18630	21631	22019
Overhead (%)	30.4	30.8	21.9	31.9	30.1

5.5.5 Correlation Calculation

The last example application performs sound propagation delay estimation based on correlation calculation. This application is used to determine the relative locations of sensors. *Small object optimization*, *runtime handle check optimization*, and *pointer dereferencing* were applied to this benchmark. As shown in Table 5.6, MEMMU increases the size of the input data by 50.6%. Although the unoptimized version of MEMMU reduces the processing rate by 57.5%, the optimized MEMMU reduces the processing rate by only 7.6%. The penalties to average power consumption of both unoptimized and optimized MEMMU are no more than 0.5%.

5.5.6 Overhead of Code Size

Table 5.8 shows the increase in code size for each benchmark. On average, executables generated with MEMMU transformations are 30% larger than those directly compiled from the original source code. Nevertheless, the code size increase does not lead to flash memory size increase in current architectures because most sensor network nodes provide sufficient flash memory, e.g., the TelosB has 48 KB of program flash memory and the MicaZ has 128 KB of program flash memory. Therefore, the overhead of code size can be neglected unless the amount of code memory becomes a tight constraint. This is not expected in the near future due to the high density of floating-gate technologies such as EEPROMs and flash memory, relative to SRAM.

5.5.7 Comparisons on Different Optimization Techniques

To understand the relative benefits of the proposed optimization techniques, we compare the improvement in performance by applying these approaches individually and in combination to five benchmarks. Table 5.7 shows the execution time of the applications with unoptimized MEMMU and MEMMU augmented with different optimization techniques. "N.A." indicates that an optimization technique cannot be applied to the corresponding benchmark. For instance, *loop transformation* cannot be used for the sensor data sampling application because the program is an implicit loop that executes the next iteration only when a hardware-triggered event occurs; there is no explicit loop structure in the code that can be transformed. Note that the *runtime handle check optimization* increases the execution time of the unoptimized MEMMU for the correlation computation benchmark because this application carries out interleaved access to two arrays. Generally, *loop transformation* with *pointer dereferencing* outperforms other optimization techniques because this combination can achieve the largest reduction in the number of handle checks and address translations.

5.5.8 Compression Ratio Estimation and Probability of Memory Exhaustion

As discussed in Section 5.4.8, the division between the compressed and the uncompressed regions is based on an estimated compression ratio. Underestimating the com-

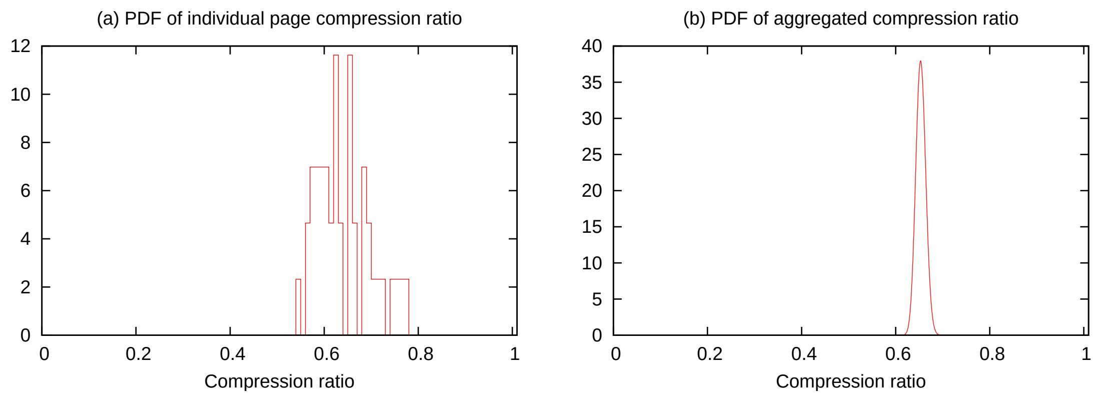


Figure 5.15: Aggregated compression ratio analysis on vibration data.

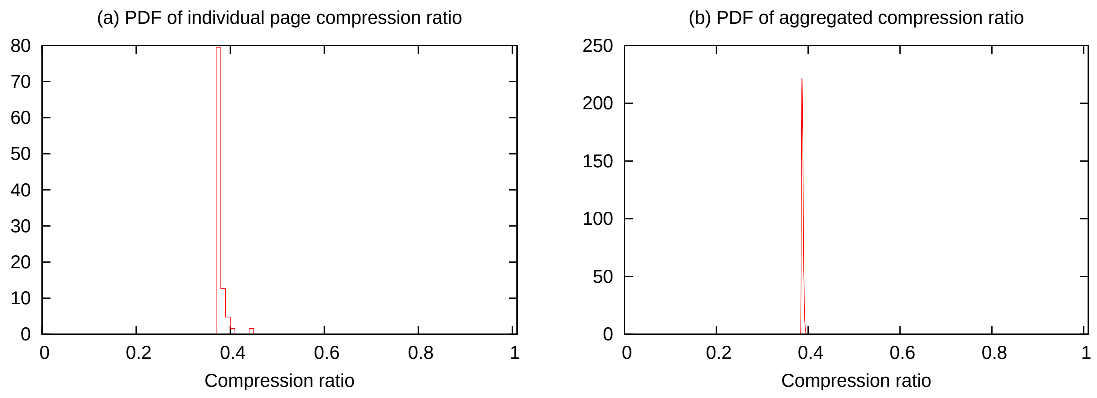


Figure 5.16: Aggregated compression ratio analysis on temperature data.

pression ratio will result in failure due to memory exhaustion. We will now use a statistical technique to analyze the probability of running out of memory for a real-world data set. The input data are vibration samples gathered from a wireless sensor network deployed in a building for infrastructure health monitoring [30]. We divide the data into 256 byte pages and compress them with the delta compression algorithm described in Section 5.4.6. The probability density function (PDF) of the page compression ratios is shown in Figure 5.15(a). The average compression ratio of an individual page is 64.7% and the standard deviation is 0.058. For a compressed region containing 30 compressed pages, we derive the average compression ratio by convolving the PDF of the page compression ratio by the number of compressed pages. Figure 5.15(b) shows the PDF of the aggregated compression ratio of pages in the compressed region. It still has an average of 64.7%, but with a much smaller standard deviation: 0.01. The standard deviation of the aggregated compression ratio decreases as the number of compressed pages increases due to the Law of Large Numbers. If we set the target compression ratio to 1.05$\times$ the average compression ratio of individual pages (i.e., 67.9%) the probability of the aggregated compression ratio exceeding our target compression ratio every time the data in the compressed region change is 0.38%. This probability drops to 1.74×10^{-6}% if we set the target compression ratio to 1.1$\times$ the average compression ratio of individual pages. If we use the data sampling period, 30 minutes, to approximate the period of updating the compressed region, Mean Time To Failure (MTTF) can be computed by dividing the sampling period by the failure probability. The MTTF increases from 131.6 hours to 2.87×10^{7} hours when we slightly increase target compression ratio from 67.9% to 71.2%. The same analysis is done with temperature data gathered from the same system. Figure 5.16 shows the results. The average compression ratio for an individual page is 38.6% and the standard deviation is 0.009. The standard deviation of the average compression ratio is 0.002. The probability of running out of memory every 30 minutes is 5.5×10^{-7}% when the estimation compression ratio is 1.05$\times$ the average. The MTTF is 9.1×10^{7} hours.

The above analysis is based on the assumption that compression ratios of pages in the compressed region are independent. Computing the correlation among pages in the compressed region is challenging and complex due to the interaction among sampling and computation. However, we can get a fairly conservative estimate of the correlation by observing that, for most applications, adjacent pages of sampled data have greater compression correlation than those that are separated by more time. We computed the correlation of compression ratios of neighboring pages, they are quite low (0.125 and

0.122) for the vibration and temperature monitoring applications.

5.5.9 Summary

To summarize, MEMMU reduces the physical memory requirements of applications by 27% or expands usable memory by up to 50%. The performance overhead of unoptimized MEMMU ranges from 57.5% to 86.3%. For four of the five benchmarks, optimization techniques reduce the performance overhead to below 10%. However, the image convolution application is an exception. Its performance overhead after optimization is 35.1% because the *pointer dereferencing* optimization technique cannot be used. There is a trade-off between memory expansion proportion and performance. Larger usable memory is obtained by using a larger compressed memory region, but this results in more compression/decompression and data migration operations, reducing speed.

Please note that we were quite conservative in our evaluation of MEMMU. The original goal of MEMMU is to expand memory allowing applications requiring more memory than physically present to still run. However, if we were to only test such large benchmarks, the outcome would often be "crash" for a system without MEMMU and "finish execution" for a system with MEMMU. Such an evaluation scheme would not illustrate the impact of MEMMU on performance. Therefore, we reduced the data set size of the application running without MEMMU and compared the data processing rates of the smaller applications with those of more demanding applications running with MEMMU.

The energy consumption overhead imposed by MEMMU depends on the duty cycle and communication activity of the applications. Duty cycle is the fraction of time that the wireless sensor mote is active. An upper-bound on the energy overhead can be derived from our average active power overhead and run time overhead. This upper-bound is 12%. Many real-world applications have duty cycles lower than 10% in order to maximize the life time of the system [49, 136]. In this case, the energy consumption overhead of MEMMU decreases as the system spends more time in idle mode. Note that the most direct alternative to using MEMMU is using a sensor network node with more RAM. This may be impossible, due to the limited types of nodes available. However, even if it is possible, increasing memory quantity increases its power consumption. An analysis with CACTI [135] indicates that for a 180 nm process, doubling the amount of memory from 10 KB to 20 KB increases read and write energy consumption by 50% and 30% respectively. Leakage power is also increased, although leakage will only be a serious problem if future sensor network node processors are fabricated using finer process technologies such as 90 nm or 65 nm. The power consumption during wireless data transmission is approximately 10× as high as when the radio is turned off for TelosB and 3.8× as high for MicaZ [107]. For applications that require periodic data transmission to a base station, or constant data exchange among nodes, the energy overhead of MEMMU will be negligible. Given 8% runtime overhead and 4% computation power overhead, Figure 5.14 shows the energy overhead of MEMMU as a function of duty cycle assuming 2% of the time is spent transmitting. For applications with duty cycles lower than 10%, MEMMU has an energy overhead smaller than 4%.

5.6 Conclusions

We have described MEMMU, an efficient software-based technique to increase usable memory in MMU-less embedded systems via automated on-line compression and decompression of in-RAM data. A number of compile-time and runtime optimizations are used to minimize its impact on the performance and power consumption. Different optimization approaches may impact performance in different ways, depending on application memory reference patterns. An efficient delta-based compression algorithm was designed for sensor data compression. MEMMU was evaluated using a number of representative wireless sensor network applications. Experimental results indicate that the proposed optimization techniques improve MEMMU's performance and that MEMMU is capable of increasing usable memory by 39% on average with less than 10% performance and power consumption penalties for all but one application. We have released MEMMU for free academic and non-profit use [87].

CHAPTER VI

Automatic Construction of System-Level Models for Sensor Networks

Rapidly and accurately estimating the impact of design decisions on performance metrics is critical to both the manual and automated design of wireless sensor networks. Estimating system-level performance metrics such as lifetime, data loss rate, and network connectivity is particularly challenging because they depend on many factors, including network design and structure, hardware characteristics, communication protocols, and node reliability. In this chapter, we describe a new method for automatically building efficient and accurate predictive models for a wide range of system-level performance metrics. These models can be used to eliminate or reduce the need for simulation during design space exploration. We evaluate our method by building a model for the lifetime of networks containing up to 120 nodes, considering both fault processes and battery energy depletion. With our adaptive sampling technique, only 0.27% of the potential solutions are evaluated via simulation, resulting in a 33.3% improvement in model accuracy compared to a uniform sampling technique. Notably, one such automatically produced model outperforms the most advanced manually designed analytical model, reducing error by 13% while maintaining very low model evaluation overhead. We also propose a new, more general definition of system lifetime that accurately captures application requirements and decouples the specification of requirements from implementation decisions.

Section 6.2 summarizes related work. Section 6.3 discusses options to obtain node-level fault models and describes the battery energy dissipation model used in our work. Section 6.4 describes the proposed technique for generating system-level models for estimating wireless sensor network performance metrics. Section 6.5 presents our automated technique for building system-level lifetime models. Section 6.5.6 compares our model with the most advanced existing analytical model. Section 6.7 concludes this chapter.

6.1 Introduction

Any sensor network design process, whether manual or automated, requires that the designer or synthesis toolchain estimate the quality of prospective designs. Many performance metrics exist, and the desirable metric is often application-dependent. The faster the metric can be estimated for a prospective design, the better, as this permits more of the solution space to be evaluated in the same amount of time. However, the estimate must also have sufficient accuracy and fidelity to support appropriate design decisions.

The modeling work in this chapter serves the goal of automated synthesis of sensor networks driven by very high-level specifications written by application domain experts. The goal of the synthesis process is to produce a sensor network implementation that meets the specifications and optimizes or bounds system-level performance metrics such as lifetime, price, and sampling resolution. Our work and related automated synthesis research [7, 19] share the need to rapidly and accurately estimate such metrics for prospective designs in the "inner loop" of the synthesis process. Accurate system-level performance models can be used to rapidly evaluate a multi-objective optimization function and find Pareto-optimal designs.

There are currently three approaches to estimating system-level performance metrics, each has a different tradeoff between efficiency and accuracy. *Measurement-based approaches* are based on data from real wireless sensor network deployments. While highly accurate, they are the most costly in terms of hardware and human effort, and are particularly challenging to use for metrics relevant to long term behavior. Measurement-based approaches are usually not used until the end of the design process. *Simulation-based approaches* are based on simulation of the prospective design. Detailed network simulation can handle numerous performance metrics but is very slow. Relying solely on simulation for design space exploration is impractical. *Analytical approaches* are based on manually constructed models that quickly compute specific performance metrics for a prospective design. However, such models are less accurate than measurement or simulation because simplifying assumptions must be made in their construction, particularly in regards to network and environment behavior. They allow rough estimation of performance metrics early during the design process, but later stages typically require other modeling techniques.

We have developed a technique for the automated construction of fast and accurate models for estimating system-level sensor network performance metrics. Our technique combines the accuracy of simulation-based approaches with the rapid evaluation time of analytical approaches. The key idea is to automatically derive a model for a system-level performance metric from measured component behavior and detailed simulation results. Model construction is done offline and may be time-consuming without cause for concern, as it is not done repeatedly during the design or synthesis process. Once the model is constructed, it can be rapidly and repeatedly evaluated.

Automated Model Construction Our technique is based on fitting a statistical model to the multidimensional observed or simulated quality metric data that characterize a design space. The black-box technique we propose can be readily automated and permits rapid evaluation of the resulting models. Numerous stochastic processes influence metrics such as system lifetime. Models constructed with the proposed process support prediction of the values of deterministic variables, and the distributions of stochastic variables. This allows a variety of metrics to be computed. In our system lifetime example, metrics such as mean time to system failure or time to n-probability of system failure can also be readily computed. As more simulation data are included, the model improves at the cost of increased model construction time. Our iterative sampling technique allows desired model accuracy to be achieved with few simulation runs. We have considered a range of alternative modeling techniques, and have found that Kriging (an interpolation method) is most appropriate [58].

Our technique also incorporates known component time-dependent characteristics into the models it builds for system-level metrics. This makes it possible to capture long-term behaviors that might not be observed in measurement or simulation that spans short time intervals. One important behavior is component failure. Node failures are common in deployed wireless sensor networks because sensor nodes are generally constructed using inexpensive components and often operate in harsh environments. However, node fault processes are often ignored when considering system-level metrics, such as lifetime. Most previous work equates node lifetime and battery lifetime. As low-power design and energy scavenging techniques are more commonly used in sensor node platforms, node-level reliability will have an increasing impact on lifetimes. In our system lifetime example, our model considers both node-level fault processes and battery depletion. We conducted experiments in which device faults were measured for a specific sensor network platform. The node temporal fault distribution we use is consistent with our measurements gathered during 21 months.

Problems with conventional definitions of system lifetime We evaluate our model construction technique using the performance metric of system lifetime, which is important for many wireless sensor networks. System lifetime has generally been defined as the duration from the start of operation until the sensor network ceases to meet its operating requirements, but most existing work uses a limited definition of "operating requirements" to simplify the system lifetime estimation problem. Past work has defined network failure as (1) first node failure [86,80], (2) first link disconnection, (3) failure of a specific number or percentage of nodes [111], and (4) disconnection of a specific number or percentage of nodes. These definitions have unfortunate implications for system design because they are often poorly related to specific application requirements. For example, the first node failure criterion is only appropriate for the rare application in which each sensor node plays a critical role.

More importantly, *lifetime metrics based on such criteria conflate specification and implementation decisions*. Consider an application in which one must sample temperature with a spatial resolution of one sample per square meter. The common metrics would not appropriately capture the lifetimes of implementations that use redundant nodes for fault tolerance because the failure of a number or percentage of nodes differs from the inability to gather data at the required spatial resolution. Coupling specification and implementation is especially troublesome if the application domain expert, e.g., a geologist or biologist, is not an expert in embedded system design. Reasoning about the relationship between network-level and application-level behaviors requires understanding the low-level system components and how they interact with each other. Domain experts rarely have the time or inclination to develop this understanding.

We believe that the definition of system lifetime should capture the requirements of application domain experts while limiting ties to implementation decisions. The definition should also be flexible enough to support a class of applications instead of a specific application. Section 6.5.2 presents and provides support for such a definition of sensor network lifetime, which can be summarized as follows: system lifetime is the duration from the start of operation until the sensor network ceases to meet the specified application-dependent but implementation-independent data gathering requirements. More generally, our automated construction process makes it possible to generate a model based on the application domain expert's preferred system lifetime metric.

Using our proposed definition of system lifetime, we applied our automatic model construction technique to modeling system lifetime for data gathering applications. Our iterative sampling technique supports construction of a predictive model with 3.6% error based on simulation of only 0.27% of the design space. With the same amount of simulation time, a uniform sampling technique derives a model with 6.0% error.

Contributions Our work makes the following contributions.

1. We are the first to propose an automatic method to construct fast and accurate models of multiple system-level metrics in wireless sensor networks. The implementation will be made publicly available.

2. We evaluate our framework by using it to build a model of system lifetime, and comparing this model with the most advanced analytic model in the literature, which it surpasses in accuracy. The resulting model itself is therefore a contribution.

3. We propose a new definition for system lifetime that better represents application requirements than current definitions and allows sensor network specification be decoupled from implementation.

4. We present a measurement-based model for node-level fault processes, and use it for system-level reliability modeling.

6.2 Related Work

Model construction from simulation or measurements with statistical methods or machine learning techniques has been used to model processor design spaces [65, 101, 27]. Previous work has demonstrated that accurate predictive models can be built by sampling a small percentage of the design space. We are the first to apply simulation-based model generation methods to sensor network system-level performance metrics. We focus on defining appropriate system-level performance metrics and developing a framework to automatically construct models to estimate them.

Researchers have previously proposed definitions and models for system lifetime [86, 111, 92]. Generally, node-level fault processes have been ignored. However, a lifetime model that considers only battery lifetime is insufficient, because node-level faults can occur before battery depletion and they also influence system performance [134, 63]. Our problem is formulated using a system lifetime definition that, as we will later argue, is more general and better suited for use by application designers. Lee et al. constructed analytical models for sensor network aging analysis using a network connectivity metric [66]. They consider node fault processes in addition to battery depletion. In contrast, we use a definition of system lifetime that decouples specification from implementation and describe a regression technique to automatically construct system-level lifetime models based on node-level characteristics. We also provide evidence that our automatically derived model is more accurate than their analytical model when evaluated using their system lifetime definition.

Node-level lifetime models can be used as a foundation for estimating system-level lifetime. Most work assumes that node lifetime equals battery lifetime, which is estimated by computing time spent in each power state [54]. A few researchers directly measured device fault processes. The developers of the ZN1 sensor node module [145] accelerated aging by inducing rapid thermal cycling in order to estimate node lifetime. Our work considers both factors, battery depletion and device faults, in order to provide more accurate estimates.

6.3 Node-Level Modeling

This section describes methods of building models for device fault processes and battery energy depletion. They are two key factors that determine the lifetimes of individual wireless sensor network nodes.

6.3.1 Fault Modeling

Sensor nodes are composed of fault-prone components. The effects of node-level faults can propagate through multiple network layers to the application level. Node-level fault models relate functionality to time, node characteristics, and node operating modes; they may be used as building blocks to estimate system-level lifetime. Models for node-level fault processes can be obtained in three ways.

1. The node manufacturer may evaluate the reliability of sensor node modules via direct testing and provide a fault model to users [145]. Models obtained in this way, however, may not characterize the in-field behavior if the deployment environment differs from the expected operating environment.

2. Node-level lifetime models may be derived from reports on prior deployments of the nodes under consideration. The more similarities between the developer's application, hardware, and deployment environment and the reference deployment, the more accurate the resulting model.

3. Finally, it is possible for application developers to experimentally characterize the sen-

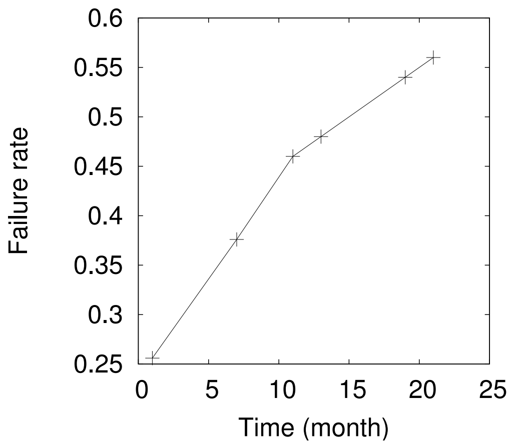


Figure 6.1: Device failure of Eco nodes.

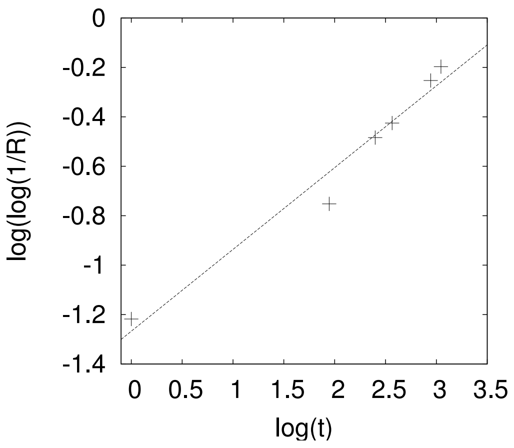


Figure 6.2: Fit failure data to Weibull distribution.

sor nodes being considered. This approach allows a controlled testing environment and workload.

We conducted experiments to model the lifetime fault distribution of ultra-compact Eco wireless sensor node [104]. The Eco node architecture consists of the nRF24E1 integrated radio and microcontroller, the Hitachi Metals H34C triaxial accelerometer, an infrared sensor, a 4 KB EEPROM, an LED, inductor, power regulator, a chip antenna, and a custom 40 mAh lithium-polymer battery. The nodes were used for various wearable applications including infant monitoring, gesture-based input devices, and water pipe monitoring. We wrote programs to test the ADC, radio, and EEPROM node components in the field, and tracked the status of 250 Eco nodes manufactured during June 2007 for 21 months.

Figure 6.1 shows the accumulated failure rate. Seven global node status evaluations were conducted during this study. Almost half of the nodes failed after 20 months. The Weibull extreme value distribution is widely used in reliability models, and is the appropriate distribution for modeling the first component fault in a node composed of many components with arbitrary temporal fault distributions [55]. We tentatively fit a Weibull distribution to the measured data. Figure 6.2 shows the log plot of time and $1/R(t)$. $R(t)$

is the reliability function. The Weibull distribution implies a linear relationship between $\ln(t)$ and $\ln(\ln(1/R(t)))$. The resulting Weibull distribution has shape parameter 0.33 and scale parameter 0.02. Its standard residual error is 0.08 and its R^2 is 0.96. The statistical significance test shows that the result is significant (p-value 0.04%). These results indicate that the measured data are consistent with those that would be produced by a fault process with a Weibull distribution. We use the resulting model in our characterization of system-level lifetime.

6.3.2 Battery Energy Dissipation Modeling

Battery models are used to predict the remaining energy of a battery and node failure time due to battery depletion. We adopt a simple battery model that assumes a constant deliverable energy capacity that is independent of variation in discharge rate. A battery is depleted when the total consumed energy equals the rated battery capacity. This model provides sufficient estimates when the battery's internal resistance and the device current are low [75]. Most sensor nodes meet these conditions. The proposed model generation technique could easily be used with more complex battery models [15].

6.4 Automatic Model Construction

This section describes our framework to automatically generate models for system-level performance metrics for sensor networks.

6.4.1 Overview

Figure 6.3 gives an overview of the automatic model construction process, which takes four types of inputs: performance metrics to be modeled (response variables), constraints on prediction error associated with the performance metrics, design parameters (predictor variables), and their associated ranges. It outputs a model for each performance metric.

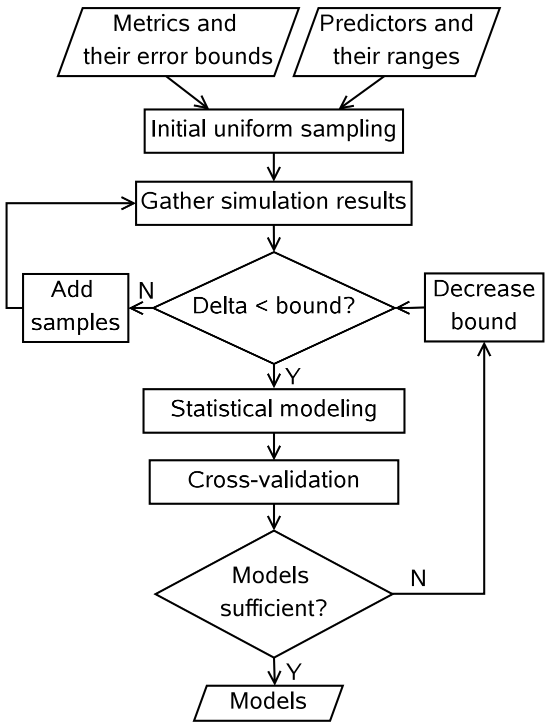
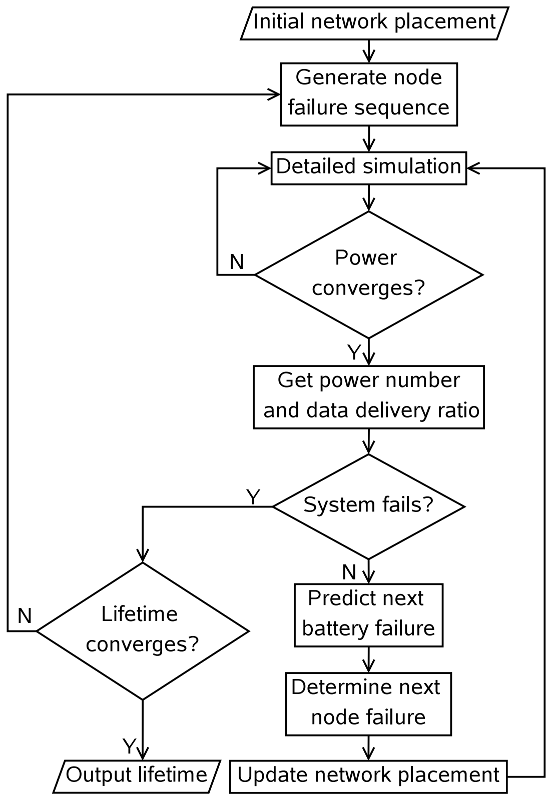


Figure 6.3: Overview of the model construction technique.



Initial network placement
Generate node failure sequence
Detailed simulation
N
Power converges?
Y
Get power number and data delivery ratio
Y
System fails?
N
Predict next battery failure
N
Lifetime converges?
Determine next node failure
Y
Output lifetime
Update network placement

Figure 6.4: Monte Carlo simulation for system lifetime distribution computation.

Our model construction technique starts with a sparse and uniformly distributed sample set. It then incrementally adds more samples in rough regions (regions where the magnitude of cost differences for adjacent points are large) according to prior simulation results. The process is iterative and contains two loops. The first loop (the one containing "add samples" in Figure 6.3) iteratively augments the sample set until differences in response variables of already sampled points that are close in the design space are below a threshold. The other loop (the one containing "decrease bound" in Figure 6.3) adjusts the bound parameter if currently derived models do not meet accuracy requirements. Each sample represents a possible value assignment to design parameters. Values of performance metrics to the samples are determined with Monte Carlo trials based on detailed sensor network simulations. Statistical modeling is used to fit the simulation results for the sampled points. Cross-validation is used to estimate the prediction error of the derived models. The procedure terminates when the estimated prediction errors meet the specified requirements. The steps in this procedure will be explained later in this section.

Our framework models multiple performance metrics simultaneously in order to reduce total simulation time. Response surfaces for different metrics may have different shapes. As a consequence, the minimum sample set required to model different metrics may differ. The model may be used by designers with different multiobjective cost functions, making it necessary to consider the surface roughness associated with each metric. However, all the metrics are modeled with the same set of samples. We choose this option for two reasons. (1) The total number of simulation runs depends on the metric that requires the largest number of samples. This technique better utilizes the available simulation results and can therefore generate more accurate models than an alternative technique using subsets of available samples to model different metrics. (2) It has the minimal implementation complexity. The only disadvantage is that model construction time for some metrics may be longer than necessary. However, since modeling is done offline, this is acceptable.

A wireless sensor network design can be evaluated with various performance metrics. We are interested in developing design tools that are accessible to domain experts who are generally not embedded system experts. To this end, we focus on system-level performance metrics that directly reflect application requirements from a domain expert's perspective. For example, domain experts may have specific requirements for end-to-end data delivery latency, but are rarely interested in node-to-node data transmission latency. System-level performance metrics such as data delivery rate, event miss rate, query response time, and unattended lifetime are affected by numerous factors. Some are specified by domain experts to characterize functionality, requirements, and the operating environment. They are fixed for the application and cannot be adjusted by design tools. Examples are size of deployment field and required sensor readings. Other factors, defined as design parameters (e.g., communication protocols, network size, and node positions) are implementation options that can be determined either manually by the designer or automatically by a design tool. The interdependencies among these factors and their complex impact on system-level performance metrics make deriving accurate closed-form analytical models for them a challenging or intractable problem.

Our technique has the following beneficial features.

1. Using a detailed sensor network simulator, allows the use of realistic simulation models, e.g., radio propagation models that consider RF signal attenuation and reflection, reception models that consider interference, or MAC protocol models that consider collisions and contention. Therefore, the design space can be modeled accurately at simulated design points.
2. Adaptive sampling and statistical modeling allows production of models that have accuracies comparable to exhaustive simulation. However, only a small part of the design space must be simulated.
3. Our technique can be used to model any system-level performance metric. Our examples consider system lifetime and data latency.
4. The constructed models can be reused and shared among numerous application developers and synthesis tools. The pool of models can be potentially expanded to support new hardware platforms or deployment environments.

6.4.2 Sampling Technique

The sampling procedure determines which design points to simulate. Using fine-grained sampling results in a long simulation time, while coarse-grained sampling results in inaccurate models. Adaptively increasing the number of samples can reduce simulation time without sacrificing model accuracy. A straightforward approach is to increase the uniform sampling resolution until accuracy requirements are met. However, this approach has significant drawbacks. Increasing the resolution for any parameter requires either invalidating all prior samples due to the new inter-sample spacing, or requires the resolution for the parameter to double. If uniform sampling is used, doubling the resolution of any parameter is very costly; even adding a single new parameter value requires m new samples, where m is the product of value counts for all other parameters. Finally, uniform sampling may introduce new samples in smooth regions of the parameter space, which will have little impact on accuracy.

We propose an algorithm that starts with sparse uniform sampling and incrementally adding samples to the rough regions. The iteration terminates when the difference in each response variable between adjacent samples is smaller than a threshold. Each iteration of the algorithm does the following. (1) For each sample point, the differences (delta) of output values between its K nearest neighbors and itself are computed. K is an empirically determined variable. (2) If the difference in output value between the sample point and any of its neighbors is larger than the given bound, a new sample is added between them. If there exists no point at the exact middle position due to discretization of some design parameters, the nearest unsimulated point is added. After normalizing each design parameter component of the vector to its range, the Euclidean distance between two samples is used to determine the nearest neighbors.

6.4.3 Modeling Technique

We consider two types of modeling methods: global polynomial regression and Kriging.

A polynomial model has the form $y = \beta_0 + \beta_1 t_1 + \cdots + \beta_m t_m + \epsilon$, where variable t_j is either a single predictor variable or a product of multiple predictors; each t_j can be raised to a positive power. ϵ is a random error with zero mean. The order of a polynomial model is determined by the maximum of the sum of the powers of the predictor variables in each term of the model. Least-squared error minimizing linear regression is used to estimate coefficients β_j.

Kriging [58] is an interpolation method that minimizes the error of estimated values based on the spatial distribution of known values. The Kriging model is defined as $y(x) = \sum_{j=1}^{N} \beta_j B_j(x) + z(x)$, where $B_j(x)$ is basis function over the experimental domain and $z(x)$ is a random error modeled as a Gaussian process. The general formula is a weighted sum of the data, $y(s_0) = \sum_{i=1}^{N} \lambda_i y(s_i)$, where s_0 is the prediction location, $y(s_i)$ is the measured value at the ith location, λ_i is an unknown weight for the measured value at the ith location, and N is the number of measured values.

The above modeling techniques are implemented in R, open-source software for statistical computing. The following functions are used in our technique: *lm* (linear regression), *Krig* (Kriging), and *cv.lm* (cross-validation).

6.4.4 Test of Model Adequacy

The prediction error of the model is estimated with 10-fold cross-validation. The sample set is randomly divided into 10 equal-sized groups. Nine are used as training data and one is used as testing data. We run the 10-fold cross-validation 50 times with different random seeds and average the results. The prediction error for a particular set of testing data is computed with the equation $E = \sqrt{\sum_{i \in T} (y_i^p - y_i^s)^2 / |T|}$, where E is the estimated error, T is the testing data set, y_i^p is the predicted value for data point i using a model constructed with the training data, and y_i^s is the simulated value for data point i. When the average error of the 50 tests is smaller than the required maximum error, we deem the model adequate.

6.4.5 Wireless Sensor Network Simulation

We use the SIDnet-SWANS simulator [43]. SWANS [12] is a scalable wireless ad hoc network simulator built on top of the JiST platform, a Java-based discrete event simulator [11]. SIDnet-SWANS extends SWANS to provide runtime interactions, integrated energy consumption modeling and management, and event monitoring facilities. Users have the flexibility to choose between different radio models, routing protocols, and MAC protocols. The energy model and packet delivery monitoring functionalities are particularly useful for the lifetime modeling presented in Section 6.5.

Note that our model construction framework can be used with any sensor network simulator. The accuracies of derived models depend on the accuracy of the simulator in use. The evaluation of the accuracy of the SIDnet-SWANS simulator is presented in Section 6.6.

6.5 System Lifetime Modeling

This section describes the use of the proposed technique to generate a model of system lifetime.

6.5.1 Domain of Applications and Assumptions

Sensor network applications span a wide domain. Different applications may have very different goals (e.g., data collection vs. object tracking) as well as different performance metrics (e.g., data delivery rate vs. even miss rate). Building one model for each specific application is infeasible since there are numerous applications. We therefore propose to divide the application domain into classes with shared characteristics. In order to select a class of application for which to generate a system lifetime model, we start with the most frequently encountered type of application (Archetype 1 identified in Chapter III): periodic data gathering in a stationary network. Applications in this class are common in environmental monitoring, infrastructural health monitoring, agriculture, and other domains. We evaluate our model generation technique for this class of applications. Note that the proposed technique is general enough for use in other domains. More detailed assumptions regarding the applications are listed in this section. Relaxing the assumptions only requires changing the simulated programs. (1) Sensor nodes are homogeneous and have the same lifetime fault model. (2) Sensor node temporal fault distributions are modeled by independent Weibull processes. (3) Sensor nodes are uniformly distributed in a 2D field. (4) A node failure disconnects the affected node from the network. (5) Data from the network are gathered at a sink node located in the center of the field. (6) Data from sensor nodes are routed to the sink using a dynamic data gathering tree. When a parent node fails, its children select other nodes in their communication range with the minimum hop count from the root node as their new parent nodes. (7) We consider two data aggregation cases: perfect aggregation and no aggregation. In the case of perfect aggregation, a single unit of data is transmitted up the routing tree regardless of the number of units of data received from children. In the case of no aggregation, each node transmits a quantity of data equal the sum of received and sensed data quantities.

6.5.2 System Lifetime Definition

We define *system lifetime* as the time elapsed since the start of operation until the spatial density of promptly delivered data drops below a threshold specified by the application developer. It allows developers to view the system from a data-oriented perspective relevant to their application requirements, while ignoring implementation details such as

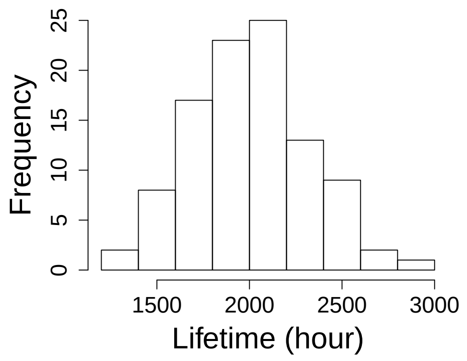


Figure 6.5: Histogram of lifetime.

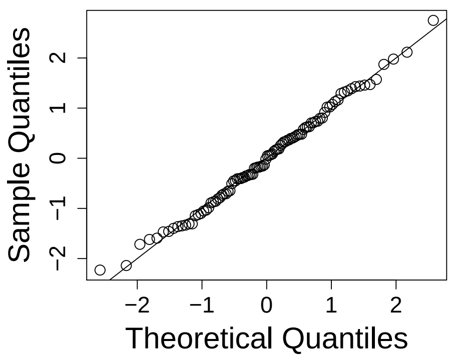


Figure 6.6: Quantile-Quantile plot of lifetime.

network structure, communication protocols, and use of redundant nodes. For example, to monitor a field with a large amount of spatial variation in data, the developer may require a higher sampling density. The sampling density criterion cannot be represented with or trivially mapped to other existing criteria. For example, the percentage of functioning nodes or the percentage of connected nodes alone cannot determine the density of data acquisition, because they do not indicate network size, network structure, and packet drop rate.

6.5.3 Predictor and Response Variables

The system lifetime of a sensor network is affected by many factors, including sensor node reliability, total number of nodes, node positions, node activities, network protocol, battery capacities, power consumptions of components in different power states, etc. The two key criteria for selecting design parameters are impact on performance and variance. Parameters that do not impact system performance or are constant should be omitted.

As a case study, we will build a lifetime model for a specific type of hardware platform and assume an outdoor deployment environment. Consequently, some parameters can be assumed to be fixed, e.g., the radio communication model parameters and the parameters of the node lifetime distribution. The proposed technique can be used to build system-level models for various hardware platforms by adjusting the appropriate simulation parameters. Six design parameters are evaluated during simulation: sampling period, network size, distance between adjacent nodes, battery capacity, aggregation, and threshold for desired data delivery density. The predictor variables are independent. They can be separately controlled without affecting each other. However, their impacts on system lifetime are interdependent. We focus on a sub-region of the design space that contains most previously deployed applications. The sub-region further determines the range of each design factor: network size ranges from 9–121 nodes; sampling density threshold ranges from 27–1000 samples per square kilometer; sampling period ranges from 10 minutes to 1 hour; and inter-node distance ranges from 100–500 feet.

For a specific network design, the system lifetime is best described using a distribution. The network may fail at different times depending on the failure times of individual nodes. Modeling lifetime with a single number, such as mean time to failure, is unnecessarily restrictive. Using a distribution within the model allows application developers to specify confidence levels for lifetime lower bounds.

The Monte Carlo simulation results suggest that system lifetime has a Gaussian distribution. Figures 6.5 and 6.6 show the histogram and the quantile–quantile plot of the lifetime for a specific network setting. Results of other network settings show a similar trend and were verified with statistical tests (the average p-value is 0.54 for tests on lifetimes of 100 network settings). We therefore assume a Gaussian distribution. We further tested our hypothesis with normality tests (a type of goodness-of-fit test that indicates whether it is reasonable to assume that random samples come from a normal distribution). According to the test results, we can accept null hypothesis that the sample data belong to a Gaussian distribution. After determining the distribution of system lifetime, two parameters are sufficient to describe it: mean and standard deviation. Our response variables are the mean and standard deviation of system lifetime.

6.5.4 Monte Carlo Simulation

For each combination of predictors corresponding to a specific network design, we use Monte Carlo simulation to obtain the system lifetime distribution. This procedure is shown in Figure 6.4. The state of the system corresponds to a particular network topology. A state change in network topology occurs upon each node failure. Each state is associated with a power profile indicating the average power consumption of each node in this state, a residual energy profile indicating the remaining battery energy for each node, and a data delivery ratio indicating the percentage of promptly delivered data. The power profile and data delivery ratio are generated using the SWANS simulator. The remaining battery lifetime of each node is then computed, allowing the time of the next node failure due to battery depletion to be estimated. The next battery depletion or node failure event causes a state change. Every time a node fails, it is removed from the network and a new network placement is generated for the next simulation run. Each Monte Carlo trial marches the system through states with decreasing node counts and data delivery ratios. Note that the run does not terminate at a user-specified data delivery ratio. Instead, sufficient data are gathered to build a model that can be evaluated for arbitrary data delivery ratios specified during model evaluation. Trials are repeated (with new, randomized, node fault failure sequences) until the mean lifetime converges.

For the sake of explanation consider Figure 6.7, which shows the result of Monte Carlo simulation for a specific network design with 49 nodes. Each line shows the degradation of data delivery ratio with time for one Monte Carlo trial. Each Monte Carlo trial starts from the same initial state, in which all nodes are operational and the residual energy of

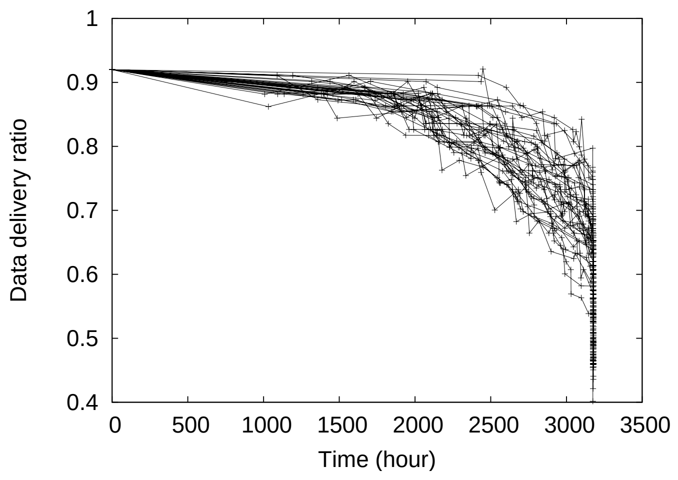


Figure 6.7: Results of Monte Carlo trials (one line each).

each node is the battery energy capacity. From this point, different Monte Carlo simulation trials, each of which is represented by a line in the figure, diverge.

If it were necessary to do prolonged network simulation for each network state, simulation time would be excessive, rendering the technique impractical. Fortunately, we observe that with a fixed network topology, the power consumption stabilizes within a few sampling periods in the simulated system. Therefore, it is not necessary to run the detailed network simulator until the next node failure. Instead, the network simulator is run long enough to determine average node power consumptions for the current network state. We found that power consumptions converge within three sampling periods for the simulated network. To be conservative, we simulated for five periods.

A python script coordinates the use of the detailed network simulator for multiple Monte Carlo trials to calculate the system lifetime distribution. Many predictor variable combinations and Monte Carlo trials are required for model construction. Therefore, we run the simulations in parallel on a cluster of machines, which is composed of over 3,500 Opteron cores. The total CPU time required for model construction was approximately 8 weeks, although the task was completed in much less time due to parallelization of the

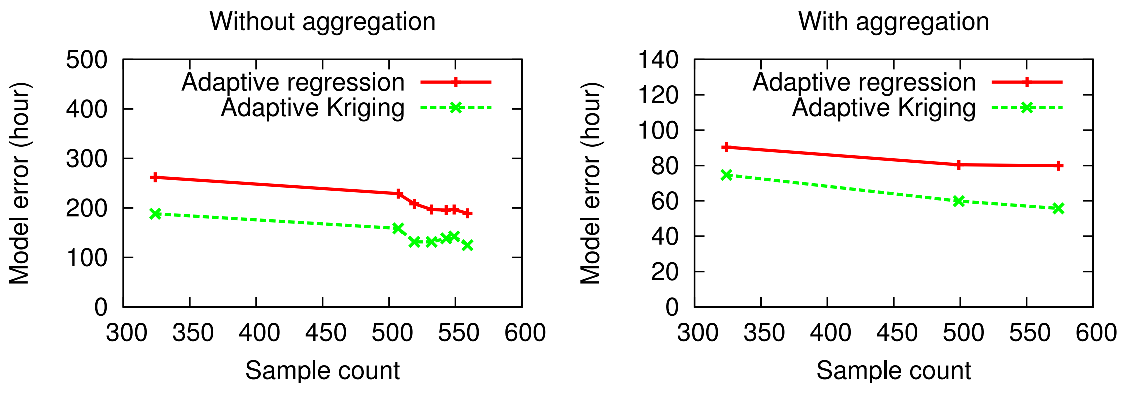


Figure 6.8: Model error and sample size.

parameter study. The model can be rapidly evaluated on a laptop computer: model use is not computationally demanding.

6.5.5 Comparison of Modeling Technique Accuracies and Efficiencies

We first compare the performance of polynomial regression and Kriging. Figure 6.8 shows the relationship between the prediction error and the sample count for applications with and without data aggregation. The x-axis represents the size of the sample set. The y-axis represents the estimated prediction error. The lines labeled “Adaptive regression” and “Adaptive Kriging” represent the errors of a 2nd-order polynomial model and a Kriging model, derived from identical sample sets determined by our adaptive sampling technique. Each point on the lines corresponds to a model generated at the end of a sampling and modeling iteration. Note that the prediction error is estimated with cross validation and is affected by how the data are partitioned. Therefore, the resulting curve is not monotonic. The errors of the polynomial regression models are always larger than those of the Kriging models. On average, the polynomial regression models have 42% larger error than the Kriging models. We conclude that Kriging is more appropriate.

The design space we consider in this case contains 405,790 potential design solutions

(31 battery capacity levels, 5 network sizes, 11 sampling periods, 17 network densities, 7 thresholds, and 2 aggregation options). Our modeling technique was able to build models with 3.6% average error (absolute error divided by average lifetime) based on approximately 1,100 simulations, i.e., 0.27% of the design space. This demonstrates that the proposed model generation technique is very efficient.

6.5.6 Comparison with an Analytical Model

To the best of our knowledge, the most relevant and most recent work is the aging analysis of wireless sensor networks by Lee et al. [66], which focuses on analyzing the degradation in network connectivity due to node-level faults and battery depletion. Their work uses a disc graph model of radio communication and ignores MAC-level behaviors, e.g., contention and collision. Unfortunately, no existing work analyzes system lifetime using our proposed definition. For the sake of comparison, we revert to a definition in past work [66], where lifetime is defined as the time until the percentage of nodes transitively connected to the sink node drops below a threshold. The our resulting model has an error of 72 hours (2.1% of average lifetime). In comparison, the average prediction error of the analytical model proposed by Lee et al. is 525 hours (15% of average lifetime).

6.6 Validation and Verification of Sensor Network Simulator

Simulator validation and verification is important because the accuracy of the simulator directly affect the accuracy of the models it is used to build, thus further affecting the optimality of design. Unfortunately, the SIDnet-SWANS simulator has not been previously validated. In this section, we test the validity of the SIDnet-SWANS simulator by comparing with real-system measurements. We also analyze the impact of uncertainty of simulation models on the system lifetime model.

Simulation validation is concerned with whether a simulator is representative of the real system. Simulation verification is concerned with whether a simulator is correctly implemented. A simulator is composed of a collection of hierarchical models. Real-system measurements is time-consuming in practice. We therefore designed our validation experiments to separately evaluate modules that are most critical to the accuracy of the system lifetime model. Since our lifetime model is mainly affected by energy consumption and network performance, the following simulation models are expected to have major effects on the quality of the system lifetime model: radio signal propagation model, energy model, and network communication model. We conducted a sequence of experiments to evaluate the simulator in terms of these metrics. Before our experiments, we have ensured that all parameters in the simulator are consistent with the numbers in the datasheets of the hardware we used for measurements.

Although SIDnet-SWANS has not been previously validated, several major assumptions or key modules used by it has been tested with other wireless network simulators. For example, the SNR-based reception model used in SIDnet-SWANS has been validated by Halkes and Langendoen [48]. The error rates for delivery ratio is lower than 5% when the SNR-based reception model is used. The IEEE 802.15.4 implementation in SIDnet-SWANS is ported from ns-2 and has been validated by Ivanov et al. [53]. The packet delivery ratio, connectivity graph, and packet latencies of ns-2 have average errors of 0.3%, 10%, and 57%.

6.6.1 Evaluate Radio Propagation Model

We start with the lowest-level simulation model that affects the network performance: the signal propagation model. The Two-Ray model is used in the simulator to compute signal attenuation from a transmitter to a receiver. This model considers a direct path and a ground reflection path from the transmitter to the receiver. When the distance d

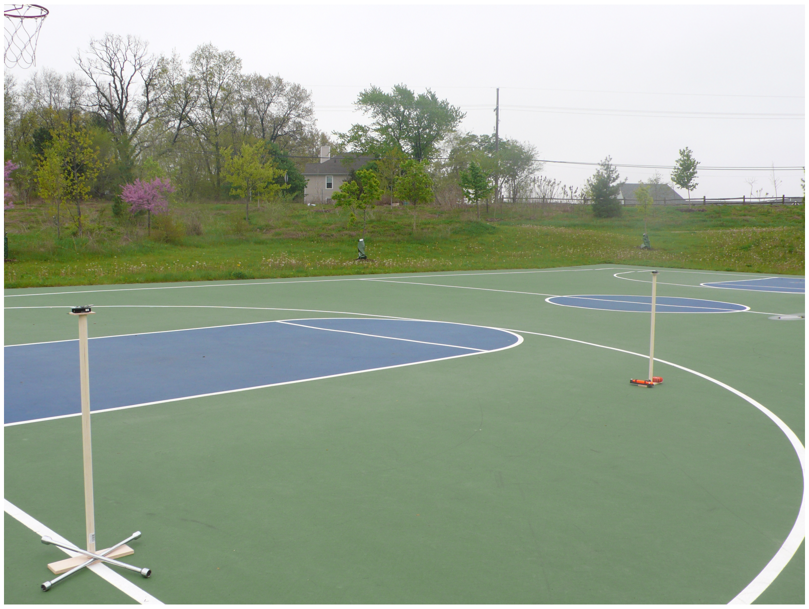


Figure 6.9: Basketball court where we validate the Two-Ray signal propagation model.

between the transmitter and the receiver is smaller than a threshold d_c (computed with Equation 6.1), Equation 6.3 is used to compute received signal strength. Otherwise, when d is larger than d_c, Equation 6.2 is used. P_t and P_r are the power of transmitted and received signals. G_t and G_r are the antenna gains of the transmitter and receiver respectively. h_t and h_r are the antenna heights of the transmitter and receiver respectively. L is the system loss. λ is the wavelength. Prior experiment with Wi-Fi channel has shown that the two-ray model fits well with the observed data in a rural environment [14].

$$d_c = \frac{4\pi h_t h_r}{\lambda} \tag{6.1}$$

$$P_r(d) = \frac{P_t G_t G_r h_t^2 h_r^2}{d^4 L} \tag{6.2}$$

$$P_r(d) = \frac{P_t G_t G_r \lambda^2}{(4\pi)^2 d^2 L} \tag{6.3}$$

To evaluate the signal propagation model, we measure RSSIs of packets transmitted

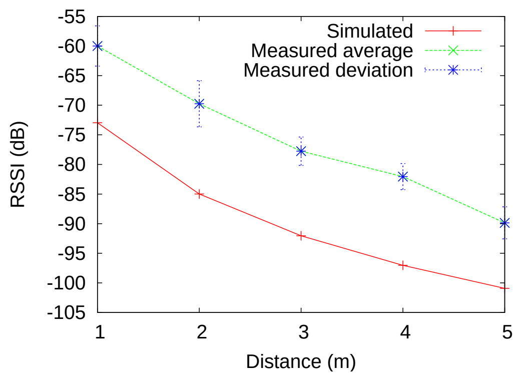


Figure 6.10: Comparison of measured and simulated RSSIs with nodes sitting on the ground.

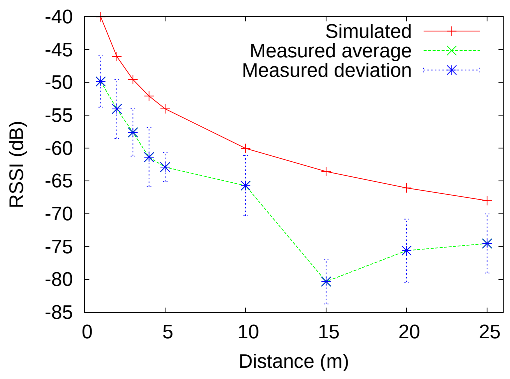


Figure 6.11: Comparison of measured and simulated RSSIs with nodes raised 0.95 m from ground.

between a pair of TelosB nodes and compare them with simulated results. We gradually increase the distance between the two nodes from 1 m to 30 m. The transmitter node sends 100 packets with 0.1 s inter-packet interval. The receiver node writes all received radio packets to its EEPROM. The nodes' orientations are fixed during the experiment. We conducted this experiment in a basketball court. Figure 6.9 shows a picture of the testing environment. Two scenarios of node placement are considered: nodes sitting directly on the ground (antenna is approximately 1.5 cm above ground) and nodes sitting on poles raised 0.95 m from the ground. We assume the gain of the receiver and the sender are both one and the signal strength of the sender is 0 dB.

Figure 6.10 and Figure 6.11 show the measured RSSIs and simulated RSSIs as a function of distance between transmitter and receiver. The maximum distance in Figure 6.10 is 5 m because when distance exceeds 5 m, packets are mostly dropped. The results indicate that the simulation results have good fidelity compared with the measurements. When the nodes are on the ground, the simulated results are about 10 dB smaller than the measured results. However, when the nodes are 0.95 m above ground, the simulated results are about 10 dB larger than the measured results. The drop in RSSI at 15 m is due to the approximately half-wavelength phase shift between the line-of-sight and the reflected sig-

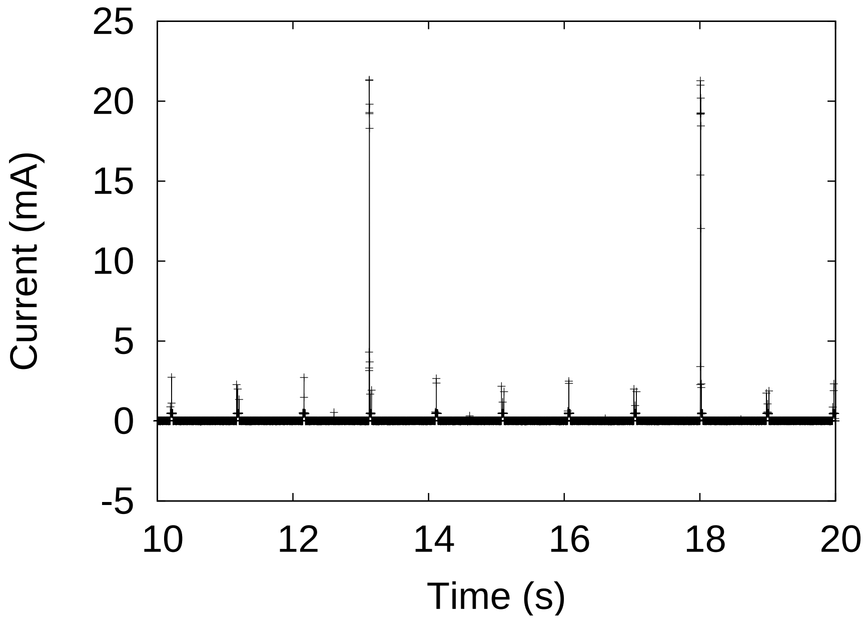


Figure 6.12: Measured power consumption.

nals. After compensating the constant offset between simulated and measured results by calibrating these parameters, the error of simulation model reduces to 2.9 dB on average.

6.6.2 Evaluate Energy Model

To evaluate the energy model, we measure the power consumption of a TelosB node while it periodically broadcasts and compare measured power consumption with simulation results. The setup for the power measurement is the same as described in Section 5.5. Figure 6.12 shows the measured current for two periods. The average power consumption is 0.19 mW. The simulated average power consumption is 0.21 mW, differing from the measurement by 10.5%.

6.6.3 Evaluate Network Communication Model

We set up an one-to-one communication experiment to measure latency of network stack in absence of contention or collision. We program two TelosB nodes to communication in a ping-pong fashion: a node sends a packet back when it receives a packet from

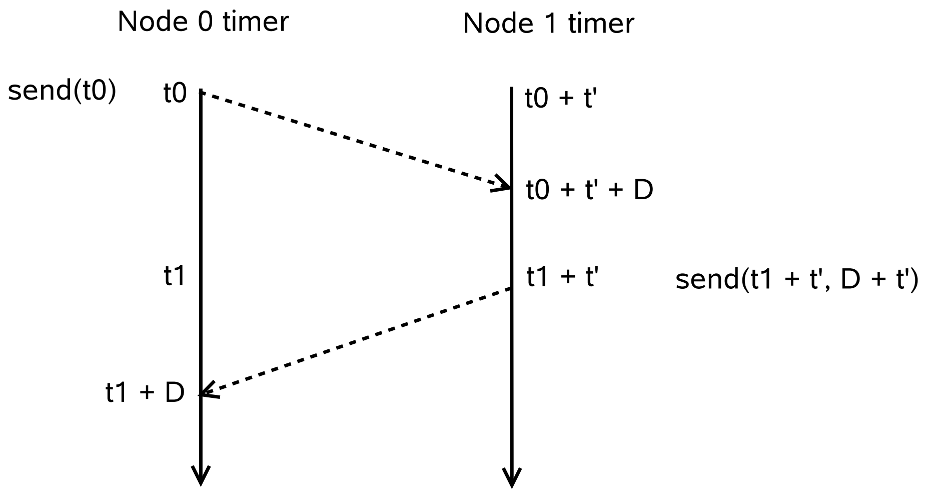


Figure 6.13: Experiment to compute latency of network stack.

another node. Each node time stamps its sent and received packet at application layer in order to compute the packet latency. Figure 6.13 demonstrates how the network latency is computed in presence of timer difference between two nodes. The two vertical arrows represent time axes of node 0 and node 1. Assume the timer of node 1 is ahead of that of node 0 by t'. In the following discussion, the word "time" refers to the local time of the node where an event occurs. At time t_0, node 0 sends a packet including the sending time t_0. t_0 is the local time of node 0 when the send function is called at application layer. Assume the total latency to transmit a packet is D. Then when node 1 receives node 0's packet, its timer reads $t_0 + t' + D$ (propagation time is ignored because it is extremely small relative to network delay). By subtracting the sending time from the receiving time, node 1 gets $t' + D$. When node 1 sends a packet back to node 0 at time $t_1 + t'$, it subtracts $t' + D$ from the current time and time stamps the packet with the result, i.e., $t1 - D$. When node 0 receives the packet from node 1, its timer indicates $t_1 + D$. By subtracting the time stamp of the packet from the receiving time, i.e., $t_1 + D - (t_1 - D)$, it gets $2D$. Therefore, the value of D can be calculated at node 0. In our experiment, we repeat the transmission for 1000 times.

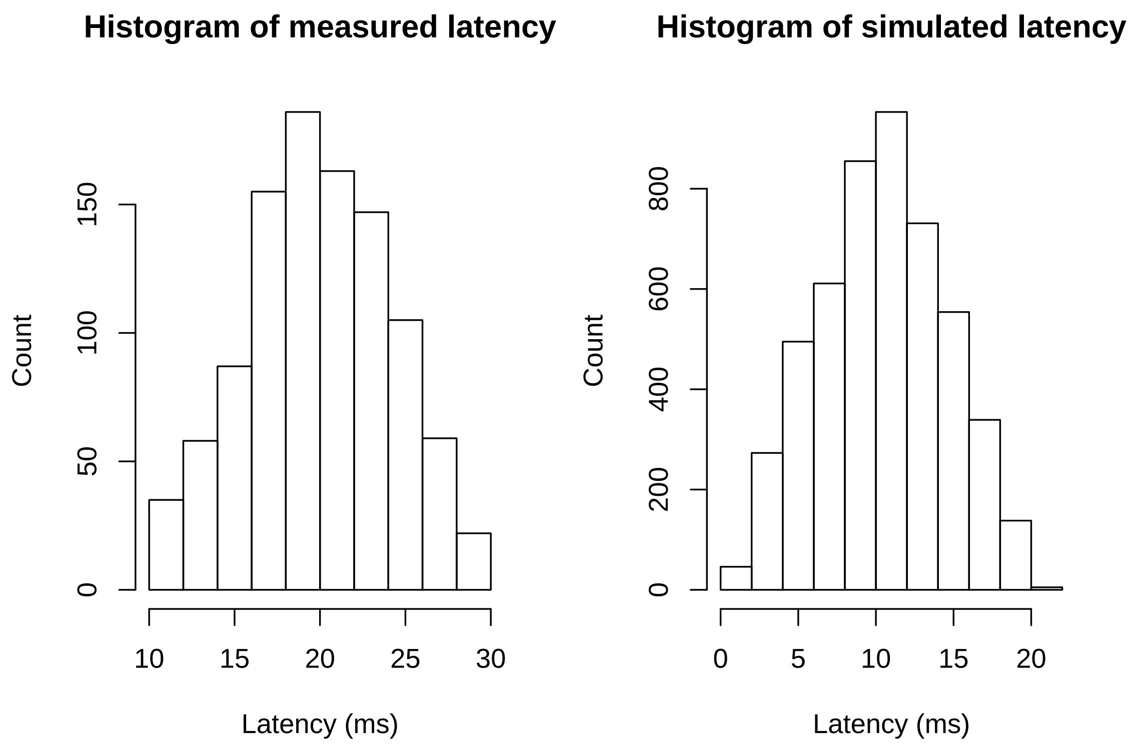


Figure 6.14: Measured and simulated network latency.

TinyOS and SIDnet-SWANS implement different CSMA backoff policies: the simulator implements a truncated binary exponential backoff policy which complies with the standard IEEE 802.15.4, while TinyOS uses a constant congestion window size. In addition, the initial and congestion backoff time window in TinyOS are larger than the backoff period in the IEEE802.15.4 protocol. For fair comparison, we modified the simulation code to conform with the implementation in TinyOS.

Figure 6.14 compares the measured and simulated packet latency. Note that the measured latency is the sum of latencies of two packets transmitted back and forth between a pair of nodes, as explained in Figure 6.13. Theoretically, the latency for one packet should have an uniform distribution, because the initial backoff is generated randomly in a range from 0.32 ms to 10.24 ms. Therefore, the total latency for two packets is expected to have a

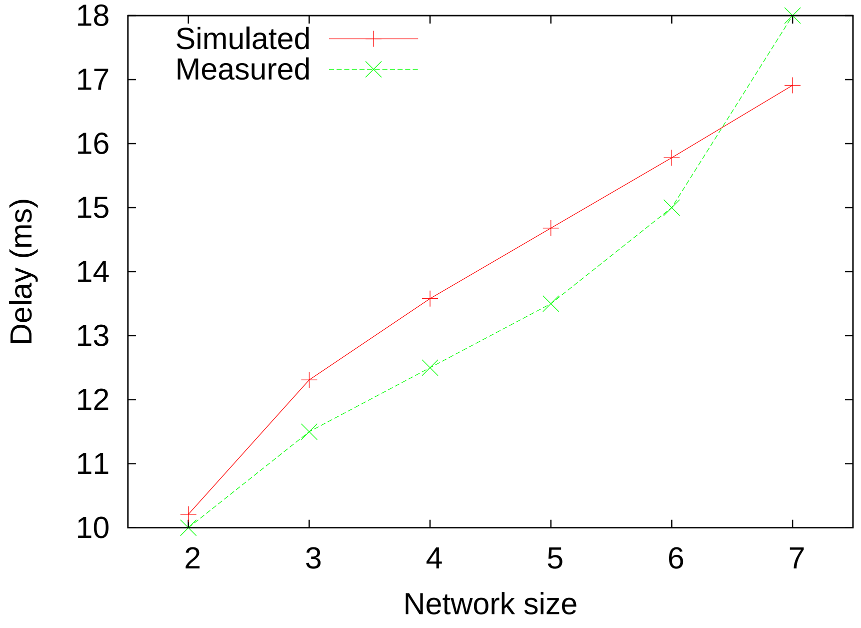


Figure 6.15: Comparison of measured and simulated packet latency.

distribution with triangle shape, which is consistent with the measured results. Figure 6.14 shows that the distributions of simulated latency and measured latency both have a triangle shape with similar width. The only obvious difference is the mean: the mean of measured latency exceeds the mean of simulated latency by 10 ms. We believe this is because the simulator does not model the execution time in the processor. Fortunately, the results indicate that this error can be easily compensated by adding 5 ms to the simulated latency of a packet.

After calibrating the simulator based on the one-to-one transmission test, we further test one-hop networks with increasing number nodes to evaluate the simulation accuracy in presence of packet contention and collision. We use the same program is as that for the one-to-one communication experiment. Figure 6.15 shows the measured and simulated 50% percentile of packet latency for different network sizes. The RMSE (root mean square error) of simulation is 1 ms.

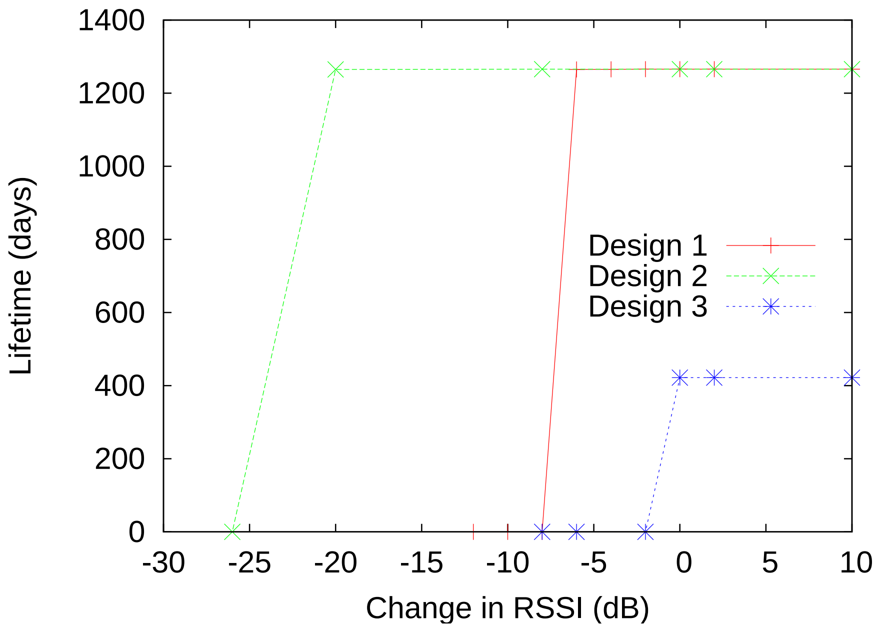


Figure 6.16: Sensitivity analysis of RSSI on lifetime model.

6.6.4 Sensitivity Analysis

From the simulation validation experiment, we observe that some parameters modeled by the simulation can be calibrated with a constant offset (e.g., packet latency), while some other parameters cannot be simply compensated with this means (e.g., RSSI). This is constrained by the detail level of a simulator as well as the inherent uncertainty in real systems. Considering all factors in the modeling technique is costly and usually unnecessary. However, it is important to understand the impact of uncertainties in certain factors on the system-level models. In this section, we take the system lifetime model as an example and study how it is affected by changed in RSSI.

We analyze how the system lifetime changes with RSSI. This process is called "sensitivity analysis". The simplest form of sensitivity analysis is to vary one value in the model by a given amount, and examine the impact that the change has on the model's result. This approach is called "one-way sensitivity analysis". In specific, we vary the RSSI computed by the SIDnet-SWANS simulator ranging from -2 dB to 26 dB while fixing other design parameters, including battery capacity, network size, network density, transmission power. For each RSSI offset, we run the MC simulation (described in Section 6.5.4) to measure the average lifetime.

We observe that the effects of RSSI change on average lifetime can be different for different designs. Figure 6.16 shows the results for three design points. X axis shows the change in RSSI. Y axis shows the average lifetime. The three lines correspond to three designs. Design 1 has a battery capacity of 3000 mAh, sampling period of 30 min, network size of 25, distance as 50 m, threshold of 8000 km^2, and transmission power of 0 dB. Design 2 has the same parameters with Design 1 except that its distance is 10 m. Design 3 differs from the Design 1 only in battery capacity and transmission power; its battery capacity is 1000 mAh and tranmission power is -5 dB. The results of these three settings have a similar pattern: the changes in lifetime with RSSI is abrupt. At a certain point, the lifetime abruptly drops to zero. In the extreme cases, the lifetime can either stay unchanged when changes in RSSI range from 2 dB to -26 dB (Design 2), or changes dramatically when RSSI decreases by 2 dB (Design 3).

This observation can be explained with principles of the radio. A signal can be successfully received when the SNR is higher than the radio reception threshold. Therefore, the different between the average RSSI of wireless links in a network determines how sensitive it is to changes in RSSI: the bigger the difference is, the bigger change in RSSI is required to bring the SNR across the radio reception threshold. For Design 2 in Figure 6.16, the distance between adjacent sensor nodes are small (10 m). The network therefore has strong links with RSSI values at least 26 dB above the radio reception threshold. Meanwhile, Design 3 uses a smaller transmission power (-5 dB) for a sparser network, so it has intermediate wireless links more prone to degradation in RSSI. The link quality directly affects the data delivery rate, which determines whether a network is still alive.

The large sensitivity of certain design to RSSI implies that if our lifetime model is directly used in design optimization, a design with a few years of expected lifetime may fail to operate immediately after deployment. We propose three solutions to address this problem. (1) Replace the signal propagation model in the simulator with a statistical model (e.g., log-normal shadowing model [113]) that is able to characterize variation of RSSI in a real system. This approach, however, may require a laborious process to collect a large number of measurements from deployment environment to compute model parameters, such as the path loss exponent and standard deviation for the log-normal shadowing model. Use a statistical model also implies longer model construction time. (2) Use guard-banding in modeling process. For example, if the RSSI can vary up to 5 dB in a real-world environment, then add a -5 dB offset to the RSSI computed by the simulation model. This approach, however, may result in suboptimal designs. (3) Consider sensitivity to undeterministic factors during optimization.

6.7 Conclusions

This chapter has described an automated technique for generating system performance models for wireless sensor networks, and explained its use to build a system lifetime model for distributed, periodic data gathering applications. We have also proposed a system lifetime definition that captures application-level requirements and decouples specification and implementation. It considers battery lifetimes and node-level fault processes. The proposed adaptive sampling technique allows the generation of lifetime models with only 3.6% error, despite simulating only 0.27% of the solutions in the design space. Taking advantage of more realistic models in sensor network simulators and offline model construction, our modeling technique reduces error by 13% compared with the most advanced analytical model, while supporting rapid model evaluation. Our modeling technique can be applied to other performance metrics. We must comment that the effectiveness of the proposed technique relies on the ability to accurately estimate relevant quality metrics for a number of potential designs, either through simulation or measurement. Our simulator validation results show that most components of the simulator are accurate or can be easily calibrated except for the signal propagation model. This problem will be addressed in Chapter VII. In Chapter VII, we will use this modeling technique in automated design of sensor networks.

CHAPTER VII

Synthesis for Wireless Sensor Networks

In Section 3.3, we have formulated the sensor network design problem as a multi-objective optimization problem. The goal of synthesis is to find the optimal sensor network design according to high-level design requirements. In this chapter, we propose and evaluate a synthesis technique for homogeneous environments. The proposed technique uses the system-level performance models presented in Chapter VI to explore the design space. We also evaluate a heuristic search for optimal design based on online simulation. The model-based design optimization is based on the assumption that the environment can be accurately modeled prior to deployment. We demonstrate this to be true for a certain type of environments categorized as homogeneous environments. Finally, we discuss challenges introduced by heterogeneity in the environment and propose possible solutions for synthesis for this more complex type of environments.

7.1 Introduction

A sensor network designer usually has a handful of system attributes to optimize. Changing a design parameter (e.g., network size) may have opposite effects on different attributes: improving some attributes and degrading some others. One challenge is to quickly identify the best design among a large number of prospective designs. Manually solving this problem requires knowledge of various existing sensor network design techniques and the interplay among system components. This is currently done by embedded system experts based on their experience. Synthesis is an automated process to refine the application-level problem specification to a more detailed specification of the system by determining values for various design parameters. It has the advantage of exploring a large set of prospective solutions to achieve good design quality.

The system-level performance models presented in Chapter VI allow quick evaluation of a prospective design. In this chapter, we propose to use these models to explore the design space during synthesis. This approach can be applied to different design problem instances and guarantee optimality. The feasibility of this approach relies on two factors: synthesis time and model accuracy. The formal determines whether the optimization process can be completed in a practical amount of time. The latter determines whether the optimal design computed based on the models is close to optimal solution in the real world.

Wireless sensor networks are greatly affected by their deployment environments. While certain types of environments can be characterized by simple models with a few parameters, more complex environments require more complex models or extra characterization effort prior to deployment. Therefore, we consider two types of deployment environments: homogeneous and heterogeneous, and separately discuss synthesis solutions for them.

7.2 Related Work

This section reviews previous work in the field of wireless sensor network optimization. Previous work has focused on formulating and solving problems of determining specific design parameters to optimize a sensor network design. Different problem formulations have been proposed and studied considering different set of costs and constraints. Researchers have studied sensor placement problem in various forms with goals to optimize network connectivity, energy consumption, data fidelity, costs, etc. [76, 84, 61]. An extensive summary on this topic is provided in Younis and Akkaya's survey [147]. Other researchers have studied problem of selecting optimal node or network configurations such as radio transmit power [103], battery allocation [79], network configuration [21], role assignment [16], routing protocol [80].

These authors only consider a limited number of design parameters and costs at one time. In addition, they assume specific optimization objectives and constraints. Therefore, the algorithms are not applicable for general purpose design problems. In contrast, we start by identifying a complete set of application-level costs that are of interests to application experts in order to solve a more general form of the design problem.

Automated design and synthesis for wireless sensor networks have been proposed by a few researchers since 2002 [8, 7, 19]. Bakshi et al. first proposed a methodology for automatic synthesis of application-specific sensor networks, based on high-level performance modeling, multi-granularity system simulation, and model refinement [8]. They focused on a synthesis form that takes a task graph as input and generates allocation and scheduling of tasks among different nodes and node settings, given constraints on latency and throughput. Bonivento et al. proposed a platform-based design method for control applications [19]. They proposed three abstraction layers representing application level, network level, and node level. Their synthesis algorithm maps algorithm and communication protocol to physical nodes to optimize energy consumption. Other researchers focus on adaptive design. Munir and Gordon proposed a dynamic optimization technique to tune sensor node parameters (processor voltage, processor frequency, and sampling frequency) online to meet application requirements [96]. The online optimization problem is formulated as a Markov Decision Process. The authors assume that an application manager calculates node-level requirements and reward function parameters based on application requirements and profiling statistics. It is not clear how this could be done.

7.3 Homogeneous and Heterogeneous Environments

Deployment environment usually has a significant impact on the behavior of a sensor network. Locations and materials of objects and external noise (e.g., Wi-Fi traffic) in the environment largely affect the signal propagation, thus impacting network performance. There are been a large number of studies that deployed wireless nodes in different environments to study characteristics of wireless channels. Their results establish evidence of environmental dependence. Researchers have measured physical layer and link layer parameters such as signal strength and packet receive rate in different environments, e.g., office building, parking lot, and natural habitat. They found that external noise, packet loss distribution, extent of "gray area", and packet delivery spatial correlation vary across different environments [148, 20, 126]. Failing to take deployment environment into consideration during sensor network design can lead to mal-functioning networks [63]. We classify deployment environments into two groups: homogeneous environment and heterogeneous environment. In a homogeneous environment, the path loss and channel conditions are independent of locations. In a heterogeneous environment, channel conditions vary from location to location. This difference determines whether an optimization technique that relies on pre-characterized system-level performance models is feasible.

We belive different methods are required in designing a sensor network for a homogeneous and heterogenous environment. Homogeneous environments can be adequately modeled with a few parameters. Therefore, a model-based design approach is proposed for this type of environment. In contrast, since general simulation models do not account for location-dependent environmental traits, such as objects in the environment, we do not expect a model-based design approach to perform well in complex heterogeneous envi-

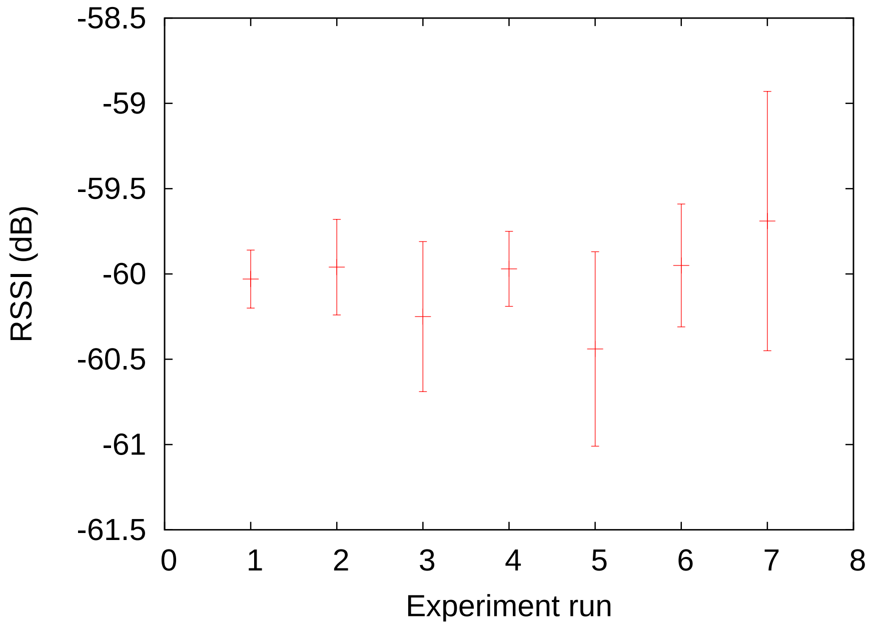


Figure 7.1: RSSI measurements with a pair of nodes with fixed distance at different locations in a basketball court.

ronments such as indoor environment. In addition, it is intractable to model all possible environments. For heterogeneous environments, we have to either count on more sophisticated sensor network simulators to accurately model a specific environment or rely on a two-phase design approach to compensate for model errors. The former can imply a significant increase in modeling time, specification complexity, or pre-characterization time. This chapter focuses on homogeneous environment. We will briefly discuss challenges and potential solutions for heterogenous environments.

7.4 Model-Based Design Optimization for Homogeneous Environments

Mostly homogeneous environments are not uncommon in the real world. An environment can be treated as homogenous environment if a significant amount of total variation in path loss and channel condition can be explained by location independent factors. Open and mostly flat fields are likely to have this property. A basketball court (at which we conducted a few experiments described later) is a good example of a homogeneous environment. We evaluate its homogeneity by measuring RSSIs of wireless links with fixed length at different locations in the field. We divided the 20×25 m basketball court into a 5×6 grid. We randomly selected seven grid edges at which we placed two TelosB nodes, one on each end of the edge. The nodes are both on poles raised 0.5 m from ground. We let one node transmit 100 packets to the other node at each location. To eliminate variations caused by other factors, the relative orientation between the transmitter and receiver nodes, as well as the roles of the two nodes, are fixed during the whole experiment. Figure 7.1 shows the measured RSSIs for seven different locations. The error bar shows standard deviation of a sequence of 100 RSSIs gathered at the same position, i.e., the temporal variation during 10 s. The deviation of average RSSIs at different positions is less than 1 dB.

7.4.1 Design Optimization With System-Level Performance Models

For homogeneous environments, we propose to build system-level performance models offline and use an exhaustive search to explore the design space using these models. The optimization engine enumerates all possible combinations of values of design parameters in a discretized design space, evaluates the total cost function of each feasible design, and selects the optimal solutions, i.e., designs with minimum total cost. Exhaustive search is feasible in this context because the system-level models can be evaluated quickly. Our experiment with an Intel Pentium 4 work station with 2.8 GHz CPU shows that it only takes 4 ms to evaluate one prospective design using the system-level performance models. Therefore, it takes only 27 m to enumerate all the 405,790 prospective designs in the whole design space considered. We formulate the sensor network design problem as a multi-objective optimization problem. For each system-level cost (x_i), three parameters are specified by an application expert: hard constraint (hc_i), soft constraint (sc_i), and weight (w_i). Their meanings are stated as follows. If $x_i > hc_i$, the corresponding solution is infeasible because violating a hard constraint is unacceptable. If $x_i \leq sc_i$, there is no need to further reduce x_i. Otherwise, if x_i is between sc_i and hc_i, x_i should be minimized with weight w_i. The goal is to minimize an integrated cost function:

$$total\_cost = \sum_i (x_i - hc_i) \times Inf + \max(x_i - sc_i, 0) \times w_i. \tag{7.1}$$

For performance metrics for which a larger value implies better solution, the metric variable is negated to be treated as cost, as is its associated soft constraint and hard constraint. For example, lifetime must be maximized. Therefore, if the specified hard constraint is 100 hours, the hc_i variable should be -100 hours in Equation 7.1. Note that this is handled by the design tool. The application designer still specifies positive values for performance metrics according to the original definition.

An alternative optimization technique is to use heuristic search with online simulations. A simulation-driven heuristic search runs simulations online to compute the integrated cost of a potential design. It decides which solution to explore next based on already explored solutions and knowledge learned about the design space. Instead of modeling the whole design space offline with our model-based optimization, the simulation-driven search usually learns a subset of the design space as it runs. It terminates when it thinks the optimal solution has been found, though this is not necessarily true. Before summarizing the pros and cons of these two approaches, we first use a set of design problems to evaluate a heuristic search algorithm.

The heuristic search considered is a greedy search algorithm. The greedy search algorithm starts with a random initial solution. It then evaluates its neighboring solutions and picks the one with the minimum total cost to move to in next step. If at any step a local minimum is reached, i.e., no neighboring solution is better than the current solution, the algorithm terminates and returns the current solution. Otherwise, it continues evaluating its neighboring solutions and moves to the one with minimal total cost. Greedy search algorithms are likely to be trapped in local minima. A common approach to escape from local minimum is to run search multiple times with different initial starting points. A simulation-driven greedy search runs simulations to evaluate its neighboring solutions. To reduce experiment time, we use the performance models built in Chapter VI to generate results for explored solutions for the greedy search. We run the greedy algorithm 100 times with different random seeds. For each run, we count the number of steps and necessary number of simulations to reach a local minimum, as well as percentage of near-optimal solutions among all local minima reached. A near-optimal solution is one with total cost exceeding cost of global optimal solution by no more than 5%.

Table 7.1 shows the results. The second and third columns show the average number of steps and simulations across 100 runs before reaching a local minimum for each problem instance. The fourth column shows the percentage of near-optimal solutions among the local minima. Averaged across all the problem instances, the greedy search algorithm takes 22.9 steps and 70 simulations to reach a local minimum. Among all local minima, 61% of them are near-optimal. In other words, the greedy algorithm has 61% probability to reach a near-optimal solution in one run. It therefore needs multiple runs to ensure finding a good solution with a high probability. The probability of encountering one near-optimal solution with N runs of greedy search is $1 - (1 - 0.61)^N$. When N is greater than 5, this probability is larger than 0.99. Therefore, the total simulation runs required to find a near-optimal solution with 0.99 confidence is $5 \times 70 = 350$. Given that one simulation takes about 73 min (consider the MC simulations required to model statistical variables). The corresponding simulation time is 450 h. Even running in parallel, the greedy search still requires 85 h (70 simulations), bounded by the time of the sequential search. We therefore conclude that a simulation-based greedy search is infeasible for practical use.

The simulation overhead for the greedy algorithm gives a rough upper bound on a simulation-driven search algorithm. More sophisticated heuristic search algorithms, e.g., gradient search and Simulated Annealing, may be developed to reduce the required number of simulation runs. However, even if a more intelligent search can reduce the simulation to one tenth that required by greedy search, it still requires hours of simulation time to find a near-optimal solution. Note that aforementioned experiment is based on a limited design space. When a larger design space is considered, e.g., by considering different hardware platforms, the simulation time required for online search is expected to increase faster than modeling time.

Synthesis time is only one factor that makes a model-based search more favorable. The benefit of using pre-characterized system-level models for design optimization is multi-fold. (1) The synthesis time is short and predictable. A reasonable bound on the synthesis time can be computed by multiplying time for evaluating one total cost function with number of potential designs. (2) Global optimality is guaranteed. (3) Models can be repeatedly used for different problem instances. With a simulation-driven search algorithm, every time the user changes the problem formulation, the design space need to be re-learned, since such a technique is usually concerned with the total cost instead of individual cost metrics.

There are a few situations for which online simulations would be more efficient or necessary. (1) When the feasible design space is tightly constrained. For example, a designer may have multiple tight hard constraints that eliminate a large number of potential solutions without running simulation (such as problem instance 5 and 6 in Table 7.1. (2) When the designer has a specific application that cannot be modeled with general-purpose system-level performance models. For example, when the designer has a pre-defined sen-

Table 7.1: Evaluation of a greedy search

Problem instance	Steps to local min	Simulations to local min	Percentage of near-optimal solutions
1	11	37	0.69
2	25	147	0.82
3	28	141	0.87
4	21	55	0.33
5	28	1	0.54
6	26	1	0.48
7	21	137	0.54
Average	22.9	74	0.61

sor placement that cannot be approximated with an uniform placement assumed in our model construction technique.

7.5 Discussion on Heterogeneous Environments

Model-based design optimization is not feasible for heterogeneous environments because it is impossible to classify heterogeneous environments into a few types and it is intractable to build models offline for all possible instances of environments. In this section, we first review past work on designing and analyzing wireless networks in heterogeneous environments. We then propose a solution for dealing with heterogeneous environments in the context of automated sensor network design.

There are two general approaches to model wireless networks in complex heterogeneous environments. The first relies on a complex simulator and a detailed model of the environment. The second relies on measurements and machine learning techniques. These two approaches can also be combined.

Simulators for indoor and urban wireless networks usually consider physical environment in their signal propagation model to account for effects such as signal reflection, signal diffraction, multi-path, etc. AT&T's WISE software for indoor wireless system deployment uses ray-tracing technique to predict performance of a network placement [34]. It requires building information including locations and compositions of walls, floors, and other obstructions to radio waves, in order to compute the possible ray paths from a transmitter to a receiver. Simulations for urban wireless network require locations of buildings, wall structures, hills, foliage, etc. [125, 74]. Detailed knowledge of environment helps improve prediction accuracy, however, the computational cost is generally high. In addition, it is not always practical to obtain and specify detailed and accurate information of physical environments.

Measurement based environment characterization has been proposed for evaluating performance and coverage of wireless networks [115, 24]. Robinson et al. proposed a technique to identify locations where a given performance metric meets a conformance threshold in deployed urban mesh networks using a constrained number of measurements. Their algorithm first divides the region into sectors with terrain-aware models, then uses measurements to refine the boundary estimate of each sector. Their method relies on the assumption that the metric monotonically changes in each sector. Chipara et al. proposed an approach to predict network coverage in indoor environments. They use signal strength measurements to fit a signal propagation model that captures effects of different types of walls. They developed an algorithm to automatically classify walls so users do not need to identify or specify types of walls. A method to plan and minimize measurements is not described.

7.5.1 Measurement-Based Environment Pre-Characterization

We believe that simulation-based optimization is necessary for heterogeneous environments. A model for the environment is still required. Using detailed and complex simulators for this purpose is impractical because it can lead to intolerable synthesis time.

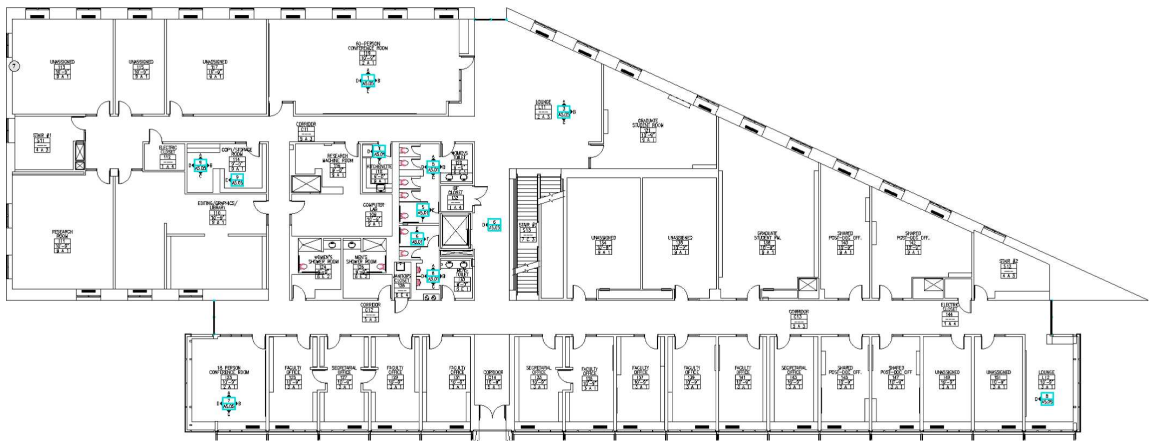


Figure 7.2: Floor plan of Motelab deployment environment.

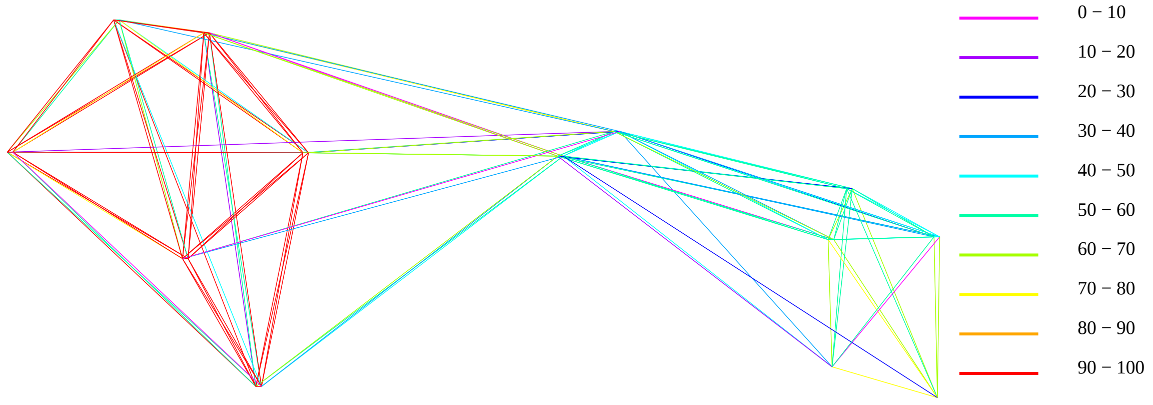


Figure 7.3: PRR of wireless links in MoteLab on first floor.

In addition, such simulators require detailed information on the environment, which is not always available to the application designer, or requires huge effort to collect and specify. An approach requiring minimal human effort and tolerable synthesis time is desired. We hypothesize that an environment can be accurately pre-characterized with few intelligently planned measurements. This hypothesis relies on one assumption: there exists strong spatial correlation in path loss and channel condition to allow using a sparse measurement to predict for locations without measurements.

To test this assumption, we evaluate how important the location information is in mod-

Table 7.2: Performance of PRR predictors.

Dataset	Network size	Total variance	MSE of dist. model	MSE of loc. model	Variance explained by location
SING-MoteLab	123	0.07	0.06	0.03	0.34
MoteLab	72	0.11	0.08	0.04	0.36

eling link qualities. Specifically, we use a neural network tool (the neural network toolset in MATLAB) to build models for PRR using measurements from an indoor environment. The neural network model maps between a data set of numeric inputs and a set of numeric targets. In our case, inputs are distances between transmitters and receivers or node positions. Outputs are packet delivery rate for wireless links. We use two-layer feed-forward network with sigmoid hidden neurons and linear output neurons. The network is trained with the Levenberg-Marquardt back-propagation algorithm. The input data set is randomly divided to three groups: training, testings, and validation, containing 80%, 10%, and 10% of the data respectively. The mean square error of the neural network model is used as a performance metric.

We apply this modeling technique to wireless link measurements gathered from the MoteLab testbed. The MoteLab testbed is deployed in a building across three floors. It is composed of 190 TMote Sky sensor motes. Each mote has a Chipcon CC2420 radio operating at 2.4GHz and is powered from wall power. Figure 7.2 shows the floorplan of the first floor. We experimented with two datasets. One is provided by the Stanford Information Networks Group [2]. In their experiments, nodes take turns to send a burst of 10,000 broadcast packets with an inter-packet interval of 10 ms. We collected the second data set from the same testbed. In our experiment, nodes take turns to send a burst of 100 packets with an inter-packet interval of 100 ms. Figure 7.3 shows measured PRR with nodes on the first floor. We can see a clear trend in spatial variation: links on the left part are much stronger than links on the right part. The spatial correlation is obvious: links close to each other are likely to have similar qualities.

The results are presented in Table 7.2. The second column shows the number of nodes involved in the experiment. At the time of our experiment, the testbed only contained 72 nodes. The third column shows the total variance of PRRs of all wireless links in the network. For each pair of nodes in the network, there exists a link. For nodes that are not connected, the link between has a PRR of zero. The fourth column shows the mean square error of the Neural Network model trained with distance between a pair of nodes as input. The fifth column shows the mean square error of the Neural Network model trained with distance between a pair of nodes as well as their locations as inputs. A node's location is described with three variables: x coordinate, y coordinate, and floor number. The last column compares the percentage of variance can be explained with the two models and shows the extra percentage of variance can be explained with node locations. On average, 35% of the variance is explained by locations. These results suggests strong spatial correlation among wireless links, which can be further used to develop a technique to use limited measurements to predict wireless link qualities in arbitrary locations in a complex environment.

7.6 Conclusions

We have described an approach for automated design optimization using system-level performance models. The system-level performance models can be quickly evaluated (4 ms for one prospective design); therefore it is practical to enumerate the whole design space during synthesis. This approach guarantees design optimality. We have evaluated an alternative that uses greedy search algorithm with online sensor network simulation. We conclude that online simulation is impractical due to long synthesis time (450 h on average). We found that complex environment imposes challenges on developing accurate models prior to the design process. We therefore categorized the deployment environment to two types and consider different synthesis approaches for them. While the model-based design optimization is useful for homogeneous environments, extra effort from application designer is required to model a heterogeneous environment accurately during the design process. We observed strong spatial correlation in an indoor heterogeneous environment. This implies that a pre-characterization technique based on a limited number of measurements may be feasible for heterogeneous environments.

CHAPTER VIII

Contributions and Conclusions

In this dissertation, we have proposed an automated design process for a class of sensor network applications and presented techniques to tackle the key challenges in supporting this process. With the proposed design process, a sensor network application designer only needs to provide high-level specifications of application functionality and requirements. The low-level implementation is automatically generated by synthesis tool and compiler. As a result, much of the human effort in the current design of sensor networks is eliminated. By hiding intricate implementation details from sensor network designers, our approach not only simplifies a designer's job and reduces programming errors, but also achieves better design quality by automatically exploring a large design space.

We initially set out the goal of developing an automated design framework for a class of sensor network applications. To address the challenges introduced by vast heterogeneity among sensor network applications, we proposed the concept of archetype-based design and categorized the application domain into seven archetypes. We attempted to select the class of applications that are the most commonly encountered, which correspond to periodic sensing and data processing from a stationary sensor network. To show that an automated design process can be accessible to individuals without embedded system design experience, we conducted a user study to evaluate designers' performance in completing the most difficult task during the design process: specifying the functionality of an application. The results of our user study show that people from other domains with little computer programming experience and no embedded system design experience can efficiently and correctly specify many representative sensor network applications using our specification language. To show that all remaining implementation can be handled by automated tools, we have designed, implemented, and evaluated compile-time and runtime techniques to generate executables from the high-level specifications. We have also designed modeling techniques and optimization techniques to determine the optimal design given designer's requirements for various system costs and performance metrics. Therefore, by developing the high-level specification languages and associated tools to generate the final implementation, we have realized the proposed automated design framework and achieved our goal.

Our work is a first step towards building a fully automated design framework for wireless sensor networks. We hope that our work will open the design of wireless sensor networks to application experts, who are not necessarily embedded system design experts. In the remainder of this chapter, we summarize the contributions of this dissertation and discuss future directions.

8.1 Contributions

We were the first to categorize sensor network application domain for the purpose of developing compact, special-purpose languages for sensor networks [130]. We identified application characteristics that affect the complexity of specification languages and generated an archetype taxonomy based on 23 existing sensor network applications. (Chapter II)

We developed a high-level language and its associated compiler for the most frequently encountered archetype [130]. Our user study indicates that archetype-specific languages have the potential to substantially improve the success rates and reduce programming times for novice programmers compared with existing general-purpose and/or node-level sensor network programming languages. Our language, WASP, increased the success rate by 1.6$\times$ and reduced average development time by 44.4% compared to other languages. (Chapter III)

We were the first to design and conduct a user study to evaluate several popular languages for sensor networks. Our user study involved 28 novice programmers and five programming languages. We have identified difficulties for novice programmers and language features to improve their efficiency and correctness. (Chapter III)

We developed a system, called FACTS, to simplify fault detection and error estimation in wireless sensor networks that is designed to be accessible to application experts [129]. We consider language features to enable novice programmers to deal with faults in sensor networks. Our technique uses easily specified domain-specific expert knowledge to support the on-line detection of some classes of sensor faults and appropriately adjust expression intervals to make the system-level impact of faults clear to sensor network users. We implemented FACTS by extending the WASP sensor network language, compiler, and run-time system. A small-scale hardware testbed and simulations of a 74-node network using real-world sensor data show that FACTS substantially increases estimation accuracy and imposes little overhead compared to fault-unaware programs. Our method can be applied to other sensor network languages. (Chapter IV)

We developed an efficient software-based technique to increase usable memory in MMU-less embedded systems via automated on-line compression and decompression of in-RAM data [131, 132]. We designed a number of compile-time and runtime optimizations to minimize its impact on the performance and power consumption. Different optimization approaches may impact performance in different ways, depending on application memory reference patterns. We also designed a delta-based compression algorithm for sensor data compression. We evaluated our technique using a number of representative wireless sensor network applications. Experimental results indicate that the proposed optimization techniques improve the performance and that our technique is capable of increasing usable memory by 39% on average with less than 10% performance and energy consumption penalties for most applications. (Chapter V)

We developed an automated technique for generating system performance models for wireless sensor networks. We focus on performance metrics that directly reflect application requirements from application experts' perspective. We developed an adaptive sampling technique to achieve desired model accuracy with few simulations. Our model construction framework supports modeling a wide range of performance metrics. (Chapter VI)

We evaluated our model construction technique by generating a system lifetime model for distributed, periodic data gathering applications [128]. We also proposed a system lifetime definition that captures application-level requirements and decouples specification and implementation. In addition to battery lifetime, we also considered node-level fault processes. The proposed adaptive sampling technique allows the generation of lifetime models with only 3.6% error, despite simulating only 0.27% of the solutions in the design space. This is a 33% improvement in accuracy over a uniform sampling technique. Taking advantage of more realistic models in sensor network simulators and offline model construction, our modeling technique reduces error by 13% compared with the most advanced analytical model, while still supporting rapid model evaluation. (Chapter VI)

We formulated the sensor network design problem as a multi-objective optimization problem and designed a specification language for designers to specify their design requirements. We found that it is feasible to find the optimal design for homogeneous environments by exhaustively exploring a huge design space using the system-level performance models. We have evaluated an alternative that uses greedy search algorithm with online simulations. We conclude that simulation-based synthesis is impractical due to long computation time (450 h on average). For more complex environments that cannot be accurately modeled prior to design time, we investigated a potential solution that uses measurements to characterize the environment. Our results suggest that the strong spatial correlation can be used in building prediction models for complex environments. (Chapter VII)

8.2 Future Work

Future work includes extending this automated design framework to support more types of applications, deployment environments, and design requirements.

8.2.1 Design Tool Evaluation for a Complete Design Cycle

This dissertation only evaluated the usability of our application programming language, which we believe is the part of the design process that an application expert is most likely to have difficulty with. Although the other required human actions such as specifying design requirements and deploying sensor nodes in the field seem simple, there may be undiscovered challenges for application experts. A user study to test how application experts engage in this design process during the whole design cycle is needed. The form of this user study is expected to be very different from the one in Chapter III: testing novice programmers with well-defined tasks in limited time. Instead, it would be useful to conduct a user study that is based on long-term interaction with real application experts during their use of our tools to develop applications in their domains.

8.2.2 Synthesis for Applications with Relaxed Assumptions

The model-based design optimization makes several assumptions about the applications, e.g., uniform node placement and homogeneous deployment environment. Offline modeling for arbitrary node placement and environment is infeasible. One potential solution is to approximate or partition the more complex design problem into simpler subproblems. Another potential solution is to use the simulation-based optimization described in Section 7.4.

In this dissertation, we focused on 2-dimensional networks. 3-dimensional placement is also common in real world and should be supported. The modeling and optimization techniques may be directly applied to 3D sensor networks given a 3D sensor network simulator as long as the computation time is tolerable.

It would be useful to embody new techniques in our design framework. This dissertation only considered battery-based power supplies. Energy harvesting is another attractive approach for supplying energy to low-power sensor nodes. Various types of energy sources exist: RF, solar, vibration, etc. An intelligent design tool should recommend the most efficient energy source based on characteristics of deployment environment and application requirements.

8.2.3 Specification Languages for Other Archetypes

To support more classes of applications, it would be useful to develop specification languages for other archetypes. The second archetype, for example, differs from the application archetype considered in this dissertation by allowing the sampling process to be triggered by certain events and by requiring the network to be interactive (the network should react to commands sent from a base station). Language features in logic programming language [17] may be useful.

APPENDIX

APPENDIX A

A.1 Measurements of Wireless Link Quality with MoteLab Testbed

In this appendix, we document an anomaly encountered in our experiments. Specifically, during our experiments to measure link quality with the MoteLab testbed, we observed that certain transmission patterns greatly degrade the measured link qualities. Although we are not able to explain the causes of such counter-intuitive observations, we did a sequence of experiments to test many hypothesis. We describe the experimental setup and results in this appendix, in the hope this anomaly can be resolved or explained in the future. Note that these problems do not affect our claims and results in the dissertation.

Section A.1.1 describes the Motelab testbed where we run the experiments. Section A.1.2 presents the simplest experimental setup for which we observe the impact of transmission pattern on PRR. Section A.1.3 describes experiments with a larger network of 79 nodes. We also justify for the experiment setup we choose to use for the dissertation work.

A.1.1 MoteLab Testbed

MoteLab is an indoor sensor network testbed deployed by researchers at Harvard University [1]. The testbed contained 72 TMote Sky sensor nodes deployed across three floors

```
event UART_Receive:
  broadcast one packet

event Radio_Receive:
  send packet to UART
```

Figure A.1: Program 1

in a building during our experiment. A TMote Sky node consists of a TI MSP430 processor and a Chipcon CC2420 radio operating at 2.4 GHz. Every node is connected to an Ethernet gateway and powered from wall power. The testbed is programmable and open to public. The web interface allows users to upload their application programs and manage their tasks. A user can reserve the whole testbed up to 30 min and select arbitrary nodes to run a program. The client program communicates with the sensor nodes in the testbed through a single controller that forwards messages to any node in the testbed and collects messages from all nodes sent through the nodes' serial ports.

A.1.2 Experiment with Three Sensor Nodes

This section describes a set of experiments with three nodes to investigate the effects of transmission patterns. We name the three nodes A, B, and C. They are all in transmission range of each other. We measure the packet reception ratio (number of received packets divided by the number of transmitted packets) of the wireless links with different transmission schedules. We experiment with different inter-packet intervals (IPI) and different transmission orders.

The programs executed on the sensor nodes are presented in pseudo code in Section A.1.2 and Figure A.1.2. The original programs are written in NesC and are compatible with TinyOS version 2.1. The difference between the two programs is how wireless transmission is triggered and how results are transmitted to a base station. The first program relies on a client program to trigger each single wireless transmission by sending a mes-

```
event UART_Receive:
  broadcast one packet

event Radio_Receive:
  broadcast one packet after time IPI
  count[sender_id] ++

event Timer_fired:
  send summarized results (count[]) to UART
```

Figure A.2: Program 2

sage to the node's serial port. The second program only requires a start command from the client program, then a sequence of transmissions are scheduled by the nodes themselves.

In the first program, as shown in Section A.1.2, a node broadcasts one packet via radio when it receives a message from its serial port. After receiving a message via the radio, a node sends a message containing information of sender ID, receiver ID, RSSI, and packet sequence number of the radio message to its serial port. A Python program runs on the client machine to send messages to nodes' serial ports to control the transmission order. The Python script also listens for messages from the nodes that report successfully received packets.

We developed the second program to minimize the impact of the control system, the network that connects the client machine and MoteLab server and associated programs that control dissemination and collection of packets, of the testbed. This program only needs one serial port message to start a sequence of wireless transmissions in a Ping-Pong fashion: nodes A and B start broadcasting when they receive a packet from each other. A delay is inserted before each transmission to control the IPI. This program also reduces use of serial port to report received packets. Instead of transmitting a message to its serial port every time it receives a radio message, a node counts the number of received packets from different senders and sends the summarized results to its serial port at the end.

Table A.1: Link Measurements with Different Transmission Orders

Transmission pattern	PRR of different links			
	A to C	B to C	A to B	B to A
AAA...BBB...	1	1	1	1
ABABAB...	1	0	1	1
BABABA...	0	1	1	1
AABBAABB...	1	1	1	1
ABBABBABB...	1	1	1	1

Table A.2: Hypothesis and Experiments

Hypothesis	Experiment
UART packets are dropped	Minimize use of UART transmission with Program 2
Gain control adjusted for one transmitter	Adjust transmission powers of A and B so that RSSI at C are the same
Affects from radio state	Restart radio after sending and receiving
Timing issue	Randomize IPI; increase IPI from 0.1 s to 10 s
Hardware problem	Switch roles of A,B,C; experiment with other nodes

Table A.1 shows the measured PRRs with different transmission orders. The results indicate that when A and B take turns to broadcast one packet, C only receives packets from the node that transmits first. Since nodes A and B both received all packets from each other, we know that the broadcasts are successful. However, in other transmission orders, C receives all packets from A and B. The results are the same when we vary other parameters such as IPI. Experiments with other configurations are listed in Table A.2. Given that they have no effects on the results, we present the results as a function of transmission order. All the experiment results are repeatable.

The same experiments are repeated with our own testbed with three TelosB nodes. The executables are exactly the same. Tmote Sky is a drop-in replacement for TelosB. They use almost the same design. All programs running on TelosB are supposed to run on Tmote Sky. However, the phenomenon shown in Table A.1 are not observed with the TelosB nodes. With all the transmission orders, node C always receives all packets from both A and B.

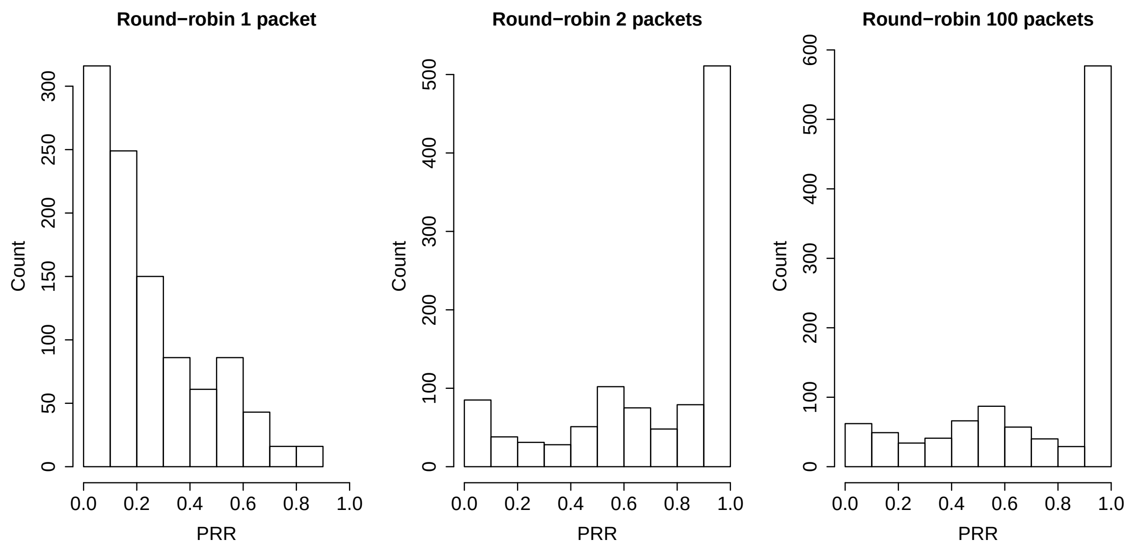


Figure A.3: Histograms of PRR with different tranmission order.

A.1.3 Experiment with All Sensor Nodes in MoteLab

Observing that the transmission pattern "ABABAB" results in broken or weak links that otherwise have good performance with other tranmission patterns, we repeated similar experiments with all nodes in the MoteLab testbed to see whether this behavior can be observed in a large network. We ran the program in Section A.1.2 with all the available nodes (72 nodes) in the testbed. We considered three transmission orders: (1) nodes take turns to send one packet each for 100 rounds, (2) nodes take turns to send two packets each for 50 rounds, and (3) nodes take turns to send 100 packets each. The IPI is fixed at 0.1 s for all experiments.

Figure A.3 shows the histograms of PRR for the three experiments. The results show that when nodes take turns to send one packet, poor and immediate links dominate. No link has a PRR higher than 90%. In the other two cases, strong links dominate. These two settings also result in similar distributions of PRR. It suggests that measurement with these two settings produce correct results.

Srinivasan et al. have done empirical study of wireless links [126]. They measured wireless link qualities in other testbeds. Although they did not claim similar observations, we found that their data show a relevant trend. Their results (Fig. 3. in [126]) demonstrate that the percentage of strong links increases with smaller IPI. With small IPIs (10 ms), they let each node send a burst of packets. For large IPIs, the nodes take turns to send every packet to reduce total experiment time. Note that also changes the transmission order in the network; we suspect this change may also have an impact on their results. Although the change in percentage of intermediate links have been explained by Srinivasan et al. as being caused by temporal correlation, the change in average link quality remains unexplained.

A.1.4 Implications on Modeling and Synthesis

Based on the observation that the local experiments with TelosB nodes does not reproduce the anomaly, we suspect that it is most likely to be a software error or a hardware artifact with Tmote Sky. We now discuss the implications of these possible causes on the automated design process.

- If it is caused by a software error either in TinyOS or the application program, it has no impact on the automated design process. With an automated design process, the low-level source code and executable are generated with tools developed and thoroughly tested by embedded system experts, which prevents such bugs from occurring in the first place.
- If it is caused by the Tmote Sky hardware, this implies that at least one extra parameter associated with the sensor node platform should be incorporated in the sensor network simulator. In other words, simulation of Tmote Sky and TelosB of the same application could produce different results on network performance. It may also

imply that a good network protocol needs to schedule network transmission to deal with the performance degradation caused by certain trait of sensor network nodes. This network protocol may decouple the network performance from the hardware feature related to our observation. In that case, the design process only needs to incorporate a different and better network protocol.

A.1.5 Conclusions

In this appendix, we have described our experiments to measure wireless link quality on the MoteLab testbed and the results that indicate the impact of transmission pattern on measured link quality. Although we are not able to explain the cause of this phenomenon, our experiments have excluded several hypothesis. Since we did not observe the same behavior with TelosB nodes, it may be a problem exclusive to the Tmote Sky platform or the MoteLab testbed. For further investigate on this problem, we would suggest conducting onsite experiments with the MoteLab testbed with direct access to the nodes or doing experiment with Tmote Sky nodes.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] moteLab project website. http://motelab.eecs.harvard.edu/.

[2] Stanford information networks group. http://sing.stanford.edu/.

[3] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, and R. Han. MANTIS: System support for MultimodAl NeTworks of In-situ Sensors. In *Proc. Int. Wkshp. Wireless Sensor Networks and Applications*, pages 50–59, September 2003.

[4] 2009. http://absynth-project.org.

[5] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: A wireless sensor network for target detection, classification, and tracking. *J. Computer Networks*, 46:605–634, 2004.

[6] David F. Bacon, Perry Cheng, and David Grove. Garbage collection for embedded systems. In *Proc. Int. Conf. Embedded Software*, pages 125–136, September 2004.

[7] A. Bakshi and V. K. Prasanna. Algorithm design and synthesis for wireless sensor networks. In *Proc. Int. Conf. Parallel Processing*, pages 423–430, August 2004.

[8] Amol Bakshi, Jingzhao Ou, and Viktor K. Prasanna. Towards automatic synthesis of a class of application-specific sensor networks. In *Proc. Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 50–58. ACM, October 2002.

[9] Amol Bakshi, Viktor K. Prasanna, Jim Reich, and Daniel Larner. The abstract task graph: a methodology for architecture-independent programming of networked sensor systems. In *Proc. Int. Conf. Mobile Systems, Applications, and Services*, pages 19–24, June 2005.

[10] Utpal Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1993.

[11] Rimon Barr, Zygmunt J. Haas, and Robbert van Renesse. JiST: an efficient approach to simulation using virtual machines. *Software: Practice and Experience*, 35(6):539–576, February 2005.

[12] Rimon Barr, Zygmunt J. Haas, and Robbert van Renesse. Scalable wireless ad hoc network simulation. In Jie Wu, editor, *Handbook on Theoretical and Algorithmic Aspects of Sensor, Ad Hoc Wireless, and Peer-to-Peer Networks*, chapter 19, pages 297–311. CRC Press, 2005.

[13] Guillermo Barrenetxea, Franois Ingelrest, Gunnar Schaefer, and Martin Vetterli. The hitchhiker's guide to successful wireless sensor network deployments. In *Proc. Int. Conf. Embedded Networked Sensor Systems*, pages 43–56, November 2008.

[14] Paolo Barsocchi, Gabriele Oligeri, and Francesco Potortì. Validation for 802.11b wireless channel measurements. Technical report, ISTI-CNR, via Moruzzi, 2006.

[15] L. Benini, G. Castelli, A. Marcii, E. Macii, M. Poncino, and R. Scarsi. A discrete-time battery model for high-level power estimation. In *Proc. Design, Automation & Test in Europe Conf.*, pages 35–39, 2000.

[16] Manish Bhardwaj and Anantha P. Chandrakasan. Bounding the lifetime of sensor networks via optimal role assignments, 2002.

[17] Urs Bischoff and Gerd Kortuem. A state-based programming model and system for wireless sensor networks. In *Proc. Int. Conf. on Pervasive Computing and Communications Wkshp.*, pages 261–266, March 2007.

[18] Surupa Biswas, Matthew Simpson, and Rajeev Barua. Memory overflow protection for embedded systems using run-time checks, reuse and compression. In *Proc. Int. Conf. Compilers, Architecture & Synthesis for Embedded Systems*, pages 280–291, September 2004.

[19] Alvise Bonivento, Luca P. Carloni, and Alberto Sangiovanni-Vincentelli. Platform based design for wireless sensor networks. *Mobile Networks and Applications*, 11(4):469–485, August 2006.

[20] Alberto Cerpa, Jennifer L. Wong, Louane Kuang, Miodrag Potkonjak, and Deborah Estrin. Statistical model of lossy links in wireless sensor networks. In *Proc. Int. Conf. Information Processing in Sensor Networks*, pages 81–88, April 2005.

[21] Yunxia Chen, Chen-Nee Chuah, and Qing Zhao. Network configuration for optimal utilization efficiency of wireless sensor networks. *Ad Hoc Networks*, 6(1):92–107, 2008.

[22] K. Chintalapudi, T. Fu, J. Paek, N. Kothari, S. Rangwala, J. Caffrey, R. Govindan, E. Johnson, and S. Masri. Monitoring civil structures with a wireless sensor network. *J. Internet Computing*, 10(2):26–34, March 2006.

[23] Krishna Chintalapudi, Jeongyeup Paek, Om Prakash, Tat Fu, Karthik Dantu, John Caffrey, Ramesh Govindan, and Erik Johnson. Structural damage detection and localization using NETSHM. In *Proc. Int. Conf. Information Processing in Sensor Networks*, pages 475–482, April 2006.

[24] Octav Chipara, Gregory Hackmann, Chenyang Lu, William D. Smart, and Gruia-Catalin Roman. Radio mapping for indoor environments. Technical report, Washington University in St. Louis, August 2009.

[25] Siddharth Choudhuri and Tony Givargis. Software virtual memory management for MMU-less embedded systems. Technical report, Center for Embedded Computer Systems, University of California, Irvine, November 2005.

[26] Thomas Clouqueur, Kewal K. Saluja, and Parameswaran Ramanathan. Fault tolerance in collaborative sensor networks for target detection. *IEEE Transactions on Computers*, 53(3):320–333, March 2004.

[27] Henry Cook and Kevin Skadron. Predictive design space exploration using genetically programmed response surfaces. In *Proc. Design Automation Conf.*, pages 960–965, June 2008.

[28] Nathan Cooprider and John Regehr. Online compression for on-chip RAM. In *Proc. Programming Languges Design and Implementation*, June 2007. To appear.

[29] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *Proc. USENIX Conf.*, pages 519–529, January 1993.

[30] C. H. Dowding and L. M. McKenna. Crack response to long-term and environmental and blast vibration effects. *J. Geotechnical and Geoenvironmental Engineering*, 131(9):1151–1161, September 2005.

[31] C. H. Dowding, H. Ozer, and M. Kotowsky. Wireless crack measurment for control of construction vibrations. In *Proc. Atlanta GeoCongress*, February 2006.

[32] Jeremy Elson and Andrew Parker. Tinker: a tool for designing data-centric sensor networks. In *Proc. Int. Conf. Information Processing in Sensor Networks*, pages 350–357, April 2006.

[33] Vadim Engelson, Dag Fritzson, and Peter Fritzson. Lossless compression of high-volume numerical data from simulations. In *Proc. Data Compression Conf.*, page 574, January 2000.

[34] Steven J. Fortune, David M. Gay, Brian W. Kernighan, Orlando Landron, Reinaldo A. Valenzuela, and Margaret H. Wright. WISE design of indoor wireless systems: Practical computation and optimization. *IEEE Computional Science and Engineering*, 2(1):58–68, March 1995.

[35] Björn Franke and Michael O'Boyle. Compiler transformation of pointers to explicit array accesses in DSP applications. In *Proc. Int. Conf. Compiler Construction*, pages 69–85, April 2001.

[36] L. Yang, Robert P. Dick, Haris Lekatsas, and Srimat Chakradhar. CRAMES: Compressed RAM for embedded systems. In *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, pages 93–98, September 2005.

[37] L. Yang, Haris Lekatsas, and Robert P. Dick. High-performance operating system controlled memory compression. In *Proc. Design Automation Conf.*, pages 701–704, July 2006.

[38] Prasanth Ganesan, Ramnath Venugopalan, Pushkin Peddabachagari, Alexander Dean, Frank Mueller, and Mihail Sichitiu. Analyzing and modeling encryption overhead for sensor network nodes. In *Proc. Int. Conf. on Wireless Sensor Networks and Applications*, pages 151–159, September 2003.

[39] David Gay, Philip Levis, David Culler, and Eric Brewer. nesC 1.1 language reference manual, May 2003.

[40] David Gay, Phillip Levis, and David Culler. Software design patterns for TinyOS. In *Proc. Languages, Compilers, and Tools for Embedded Systems*, pages 40–49, June 2005.

[41] Johannes Gehrke and Samuel Madden. Query processing in sensor networks. *Pervasive Computing*, 3(1):46–55, January 2004.

[42] Marius Ghercioiu. A graphical programming approach to wireless sensor network nodes. In *Proc. Sensors for Industry Conf.*, pages 118–121, February 2005.

[43] Oliviu C. Ghica, Goce Trajcevski, Peter Scheuermann, Zachary Bischof, and Nikolay Valtchanov. SIDnet-SWANS: a simulator and integrated development platform for sensor networks applications. In *Proc. Int. Conf. Embedded Networked Sensor Systems*, pages 385–386, November 2008.

[44] Ben Greenstein, Eddie Kohler, and Deborah Estrin. A sensor network application construction kit (SNACK). In *Proc. Int. Conf. Embedded Networked Sensor Systems*, pages 69–80, November 2004.

[45] Lin Gu, Dong Jia, Pascal Vicaire, Ting Yan, Liqian Luo, Ajay Tirumala, Qing Cao, Tian He, John A. Stankovic, Tarek Abdelzaher, and Bruce H. Krogh. Lightweight detection and classification for wireless sensor networks in realistic environments. In *Proc. Int. Conf. Embedded Networked Sensor Systems*, pages 205–217, November 2005.

[46] Carlos Guestrin, Peter Bodi, Romain Thibau, Mark Paski, and Samuel Madde. Distributed regression: an efficient framework for modeling sensor network data. In *Proc. Int. Conf. Information Processing in Sensor Networks*, pages 1–10, April 2004.

[47] Ramakrishna Gummadi, Nupur Kothari, Todd Millstein, and Ramesh Govindan. Declarative failure recovery for sensor networks. In *Proc. Int. Conf. Aspect-oriented software development*, pages 173–184. ACM, March 2007.

[48] Gertjan Halkes and Koen Langendoen. Experimental evaluation of simulation abstractions for wireless sensor network MAC protocols. Technical report, Delft University of Technology, January 2009.

[49] Carl Hartung, Richard Han, Carl Seielstad, and Saxon Holbrook. FireWxNet: a multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments. In *Proc. Int. Conf. Mobile Systems, Applications, and Services*, pages 28–41, June 2006.

[50] Joseph M. Hellerstein and Wei Wang. Optimization of in-network data reduction. In *Proc. of Int. Wkshp. on Data Management for Sensor Networks*, pages 40–47, August 2004.

[51] Timothy W. Hnat, Tamim I. Sookoor, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse. Macrolab: A vector-based macroprogramming framework for cyber-physical systems. In *Proc. Int. Conf. Embedded Networked Sensor Systems*, pages 225–238, November 2008.

[52] James Horey, Eric Nelson, and Arthur B. Maccabe. Tables: A table-based language environment for sensor networks. Technical report, The University of New Mexico, 2007.

[53] Svilen Ivanov, André Herms, and Georg Lukas. Experimental validation of the ns-2 wireless model using simulation, emulation, and real network. In *Proc. on Mobile Ad-Hoc Networks*, 2007.

[54] Deokwoo Jung, Thiago Teixeira, and Andreas Savvides. Sensor node lifetime analysis: Models and tools. *ACM Trans. on Sensor Networks*, 5(1):1–33, February 2009.

[55] Eric Karl, David Blaauw, Dennis Sylvester, and Trevor Mudge. Reliability modeling and management in dynamic microprocessor-based systems. In *Proc. Design Automation Conf.*, pages 1057–1060, July 2006.

[56] Chris Karlof and David Wagner. Secure routing in wireless sensor networks: Attacks and countermeasures. *Elsevier's AdHoc Networks J.*, 1(2–3):293–315, September 2003.

[57] Sukun Kim, Shamim Pakzad, David Culler, James Demmel, Gregory Fenves, Steven Glaser, and Martin Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *Proc. Int. Conf. Information Processing in Sensor Networks*, pages 254–263, April 2007.

[58] J.P.C. Kleijnen. Kriging metamodeling in simulation: A review. Technical report, Tilburg University, Center for Economic Research, 2007.

[59] Nupur Kothari, Ramakrishna Gummadi, Todd Millstein, and Ramesh Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *Proc. Programming Languges Design and Implementation*, pages 200–210, June 2007.

[60] Farinaz Koushanfar, Miodrag Potkonjak, and Alberto Sangiovanni-Vincentelli. Online fault detection of sensor measurements. In *Proc. Int. Conf. on Sensors*, pages 974–980, October 2003.

[61] Andreas Krause, Ram Rajagopal, Anupam Gupta, and Carlos Guestrin. Simultaneous placement and scheduling of sensors. In *Proc. Int. Conf. Information Processing in Sensor Networks*, pages 181–192, 2009.

[62] NI LabVIEW. http://www.ni.com/labview.

[63] Koen Langendoen, Aline Baggio, and Otto Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *Proc. Int. Wkshp. Parallel and Distributed Real-Time Systems*, pages 1–8, April 2006.

[64] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. Int. Symp. Code Generation and Optimization*, pages 75–86, March 2004.

[65] Benjamin C. Lee and David M. Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, pages 185–194, October 2006.

[66] Jae-Joon Lee, Bhaskar Krishnamachari, and C.-C. Jay Kuo. Aging analysis in large-scale wireless sensor networks. *Ad Hoc Networks*, 6(7):1117–1133, September 2008.

[67] Haris Lekatsas, Jörg Henkel, and Wayne Wolf. Code compression for low power embedded system design. In *Proc. Design Automation Conf.*, pages 294–299, June 2000.

[68] P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *Proceedings of Internation Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[69] P. Levis, D. Gay, and D. Culler. Bridging the gap: Programming sensor networks with application specific virtual machines. Technical report, UC Berkeley, August 2004.

[70] Philip Levis. The TinyScript language, July 2004.

[71] Philip Levis, Nelson Lee, Matt Welsh, and David E. Culler. TOSSIM: accurate and scalable simulation of entire tinyOS applications. In *Proc. Int. Conf. Embedded Networked Sensor Systems*, pages 126–137, November 2003.

[72] D. Li, K. Wong, Y. Hu, and A. Sayeed. Detection, classification, and tracking of targets. *Signal Processing Magazine*, 19(2):17–29, March 2002.

[73] Mo Li and Yunhao Liu. Underground structure monitoring with wireless sensor networks. In *Proc. Int. Conf. Information Processing in Sensor Networks*, pages 69–78, April 2007.

[74] G. Liang and H. L. Bertoni. A new approach to 3-d ray tracing for propagation prediction in cities. *IEEE Trans. Antennas and Propagation*, 46(6):853–863, June 1998.

[75] David Linden and Thomas B. Reddy. *Handbook of Batteries*. MacGraw-Hill, 2002.

[76] Hai Liu, Peng-Jun Wan, and Xiaohua Jia. *Fault-tolerent relay node placement in wireless sensor networks*, pages 230–239. Springer, August 2005.

[77] Kebin Liu, Mo Li, Yunhao Liu, Minglu Li, Zhongwen Guo, and Feng Hong. Passive diagnosis for wireless sensor networks. In *Proc. Int. Conf. Embedded Networked Sensor Systems*, pages 113–126, November 2008.

[78] Ting Liu, Christopher M. Sadler, Pei Zhang, and Margaret Martonosi. Implementing software on resource-constrained mobile sensors: experiences with Impala and ZebraNet. In *Proceedings of International Conference on Mobile systems, Applications, and Services*, pages 256–269, June 2004.

[79] Hengyu Long, Yongpan Liu, Yiqun Wang, Robert P. Dick, and Huazhong Yang. Battery allocation for wireless sensor network lifetime maximization under cost constraints. In *Proc. Int. Conf. Computer-Aided Design*, pages 705–712, November 2009.

[80] R Madan and S Lall. Distributed algorithms for maximum lifetime routing in wireless sensor networks. *IEEE J. Wireless Communications*, pages 2185–2193, August 2006.

[81] Sam Madden, Joe Hellerstein, and Wei Hong. TinyDB: In-network query processing in TinyOS, September 2003.

[82] Samuel Madden, Michael Franklin, Joseph Hellerstein, and Wei Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Trans. Database Systems*, 30(1):122–173, March 2005.

[83] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: a tiny aggregation service for ad-hoc sensor networks. In *Proc. Int. Symp. Operating Systems Design and Implementation*, pages 131–146, December 2002.

[84] Morteza Maleki. QoM and lifetime-constrained random deployment of sensor networks for minimum energy consumption. In *Proc. Int. Conf. Information Processing in Sensor Networks*, April 2005.

[85] Kathryn S. Mckinley, Steve Carr, and Chau wen Tseng. Improving data locality with loop transformations. In *ACM Trans. Programming Languages and Systems*, pages 424–453, July 1996.

[86] Duarte E. J. Melo and M. Liu. Analysis of energy consumption and lifetime of heterogeneous wireless sensor networks. In *Proc. Global Telecommunications Conf.*, volume 1, pages 21–25, November 2002.

[87] Memory expansion on embedded systems without MMUs. MEMMU link at http://www.eecs.northwestern.edu/~dickrp/tools.html.

[88] William Mendenhall and Terry Sincich. *Statistics for engineering and the sciences*. Dellen Macmillan, 1992.

[89] Jeffrey Scott Miller, Peter A. Dinda, and Robert P. Dick. Evaluating a BASIC approach to sensor network node programming. In *Proc. Conf. on Embedded Networked Sensor Systems*, pages 155–168, November 2009.

[90] Ramon Moore. *Interval Analysis*. Prentice-Hall, 1817.

[91] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks: Fundamental concepts and state of the art. Technical report, University of Trento, 2008.

[92] Laurent Mounier, Ludovic Samper, and Wassim Znaidi. Worst-case lifetime computation of a wireless sensor network by model-checking. In *Proc. Wkshp. on Performance Evaluation of Wireless Ad Hoc, Sensor, and Ubiquitous Networks*, pages 1–8. ACM, October 2007.

[93] Steven S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Publishers, 1997.

[94] Réne Müller. SwissQM query interface. http://www.swissqm.inf.ethz.ch/querygui.html.

[95] René Müller, Gustavo Alonso, and Donald Kossmann. SwissQM: Next generation data processing in sensor networks. In *Proc. Conf. on Innovative Data Systems Research*, pages 1–9, January 2007.

[96] Arslan Munir and Ann Gordon-Ross. An MDP-based application oriented optimal policy for wireless sensor networks. In *Proc. Int. Conf. Hardware/Software Codesign and System Synthesis*, pages 183–192. ACM, October 2009.

[97] Suman Nath, Phillip B. Gibbons, Srinivasan Seshan, and Zachary R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *Proc. Int. Conf. Embedded Networked Sensor Systems*, pages 250–262, November 2004.

[98] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *Proc. Int. Conf. Information Processing in Sensor Networks*, pages 489–498, April 2007.

[99] Kevin Ni, Nithya Ramanathan, Mohamed Nabil Hajj Chehade, Laura Balzano, Sheela Nair, Sadaf Zahedi, Eddie Kohler, Greg Pottie, Mark Hansen, and Mani Srivastava. Sensor network data fault types. *ACM Trans. on Sensor Networks*, 5(3):1–29, May 2009.

[100] Markus F.X.J. Oberhumer. LZO real-time data compression library. http://www.oberhumer.com/opensource/lzo.

[101] Berkin Ozisikyilmaz, Gokhan Memik, and Alok Choudhary. Efficient system design space exploration using machine learning techniques. In *Proc. Design Automation Conf.*, pages 966–969, June 2008.

[102] Jeongyeup Paek, K. Chintalapudi, R. Govindan, J. Caffrey, and S. Masri. A wireless sensor network for structural health monitoring: performance and experience. In *Proc. Wkshp. on Embedded Networked Sensors*, pages 1–9, 2005.

[103] Sooksan Panichpapiboon, Gianluigi Ferrari, and Ozan K. Tonguz. Optimal transmit power in wireless sensor networks. *IEEE Trans. on Mobile Computing*, 5(10):1432–1447, October 2006.

[104] Chulsung Park, Jinfeng Liu, and Pai H. Chou. Eco: an ultra-compact low-power wireless sensor node for real-time motion monitoring. In *Proc. Int. Conf. Information Processing in Sensor Networks*, pages 398–403, April 2005.

[105] Cristiano Pereira, Sumit Gupta, Koushik Niyogi, Iosif Lazaridis, Sharad Mehrotra, and Rajesh Gupta. Energy efficient communication for reliability and quality aware sensor networks. Technical report, University of California at Irvine, April 2003.

[106] PLY. http://www.dabeaz.com/ply.

[107] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: enabling ultra-low power wireless research. In *Proc. Int. Conf. Information Processing in Sensor Networks*, April 2005.

[108] Joseph Polastre, Robert Szewczyk, Alan Mainwaring, David Culler, and John Anderson. Analysis of wireless sensor networks for habitat monitoring. In C. S. Raghavendra, Krishna M. Sivalingam, and Taieb Znati, editors, *Wireless Sensor Networks*, chapter 18, pages 399–423. Springer US, 2004.

[109] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Commun. ACM*, 43(5):51–58, May 2000.

[110] S. Sandeep Pradhan, Julius Kusuma, and Kannan Ramchandran. Distributed compression in a dense microsensor network. *IEEE Signal Processing Magazine*, 19(2):51–60, March 2002.

[111] Vivek Rai and Rabi N. Mahapatra. Lifetime modeling of a sensor network. In *Proc. Design, Automation & Test in Europe Conf.*, pages 202–203, March 2005.

[112] Nithya Ramanathan, Kevin Chang, Rahul Kapur, Lewis Girod, Eddie Kohler, and Deborah Estrin. Sympathy for the sensor network debugger. In *Proc. Int. Conf. Embedded Networked Sensor Systems*, pages 255–267, 2005.

[113] Theodore S. Rappaport. *Wireless Communications: Principles & Practice*. Prentice-Hall, NJ, 2002.

[114] Luigi Rizzo. A very fast algorithm for RAM compression. *Operating Systems Review*, 31(2):36–45, April 1997.

[115] Joshua Robinson, Ram Swaminathan, and Edward W. Knightly. Assessment of urban-scale wireless networks with a small number of measurements. In *Proc. Int. Conf. Mobile Computing and Networking*, pages 187–198. ACM, September 2008.

[116] Kay Römer and Friedemann Mattern. The design space of wireless sensor networks. *J. of Wireless Communications*, 11(6):54–61, December 2004.

[117] Andreas Savvides, Wendy Garber, Sachin Adlakha, Randolph Moses, and Mani B. Srivastava. On the error characteristics of multihop node localization in ad-hoc sensor networks. In *Proc. Int. Wkshp. Information Processing in Sensor Networks*, pages 317–332, April 2003.

[118] Cory Sharp, Shawn Schaffert, Alec Woo, Naveen Sastry, Chris Karlof, Shankar Sastry, and David Culler. Design and implementation of a sensor network system for vehicle tracking and autonomous interception. In *Proc. European Wkshp. on Wireless Sensor Networks*, pages 93–107, January 2005.

[119] Anmol Sheth, Kalyan Tejaswi, Prakshep Mehta, Chandresh Parekh, R. Bansal, Shabbir N. Merchant, Trilok N. Singh, Uday B. Desai, Chandramohan A. Thekkath, and K. Toyama. Senslide: a sensor network based landslide prediction aystem. In *Proc. Int. Conf. Embedded Networked Sensor Systems*, pages 280–281, November 2005.

[120] Victor Shnayder, Bor rong Chen, Konrad Lorincz, Thaddeus, and Matt Welsh. Sensor networks for medical care. Technical report, Harvard University, April 2005.

[121] Pavan Sikka, Peter Corke, Philip Valencia, Christopher Crossman, Dave Swain, and Greg Bishop-Hurley. Wireless adhoc sensor and actuator networks on the farm. In *Proc. Int. Conf. Information Processing in Sensor Networks*, pages 492–499, April 2006.

[122] Gyula Simon, Miklós Maróti, Ákos Lédeczi, György Balogh, Branislav Kusy, András Nádas, Gábor Pap, János Sallai, and Ken Frampton. Sensor network-based countersniper system. In *Proc. Int. Conf. Embedded Networked Sensor Systems*, pages 1–12, November 2004.

[123] Vipul Singhvi, Andreas Krause, Carlos Guestrin, Jr. James H. Garrett, and H. Scott Matthews. Intelligent light control using sensor networks. In *Proc. Int. Conf. Embedded Networked Sensor Systems*, pages 218–229, November 2005.

[124] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: A language and runtime system for perpetual

systems. In *Proc. Int. Conf. Embedded Networked Sensor Systems*, pages 161–174, November 2007.

[125] Vinay Sridhara and Stephan Bohacek. Realistic propagation simulation of urban mesh networks. *J. Computer Networks*, 51(12):3392–3412, August 2007.

[126] Kannan Srinivasan, Prabal Dutta, Arsalan Tavakoli, and Philip Levis. An empirical study of low-power wireless. *ACM Trans. Sensor Networks*, 6:1–49, March 2010.

[127] Ivan Stoianov, Lama Nachman, Sam Madden, and Timur Tokmouline. Pipenet: a wireless sensor network for pipeline monitoring. In *Proc. Int. Conf. Information Processing in Sensor Networks*, pages 264–273, April 2007.

[128] L. Bai, R. P. Dick, P. Chou, and P. A. Dinda. Automated construction of fast and accurate system-level models for wireless sensor networks. In *Proc. Design, Automation & Test in Europe Conf.*, pages 1083–1088, March 2011.

[129] L. Bai, R. P. Dick, P. A. Dinda, and P. Chou. Simplified programming of faulty sensor networks via code transformation and run-time interval computation. In *Proc. Design, Automation & Test in Europe Conf.*, pages 88–93, March 2011.

[130] L. S. Bai, Robert P. Dick, and Peter A. Dinda. Archetype-based design: sensor network programming for application experts, not just programming experts. In *Proc. Int. Conf. Information Processing in Sensor Networks*, pages 85–96, April 2009.

[131] L. S. Bai, L. Yang, and Robert P. Dick. Automated compile-time and run-time techniques to increase usable memory in MMU-less embedded systems. In *Proc. Int. Conf. Compilers, Architecture & Synthesis for Embedded Systems*, pages 125–135, October 2006.

[132] L. S. Bai, L. Yang, and Robert P. Dick. MEMMU: Memory expansion for MMU-less embedded systems. *ACM Trans. Embedded Computing Systems*, 8(3):23–33, April 2009.

[133] Ryo Sugihara and Rajesh K. Gupta. Programming models for sensor networks: A survey. *ACM Trans. on Sensor Networks*, 4(2):1–29, March 2008.

[134] Robert Szewczyk, Joseph Polastre, Alan Mainwaring, and David Culler. Lessons from a sensor network expedition. In *Proc. European Wkshp. on Sensor Networks*, January 2004.

[135] David Tarjan, Shyamkumar Thoziyoor, and Norman P. Jouppi. CACTI 4.0. Technical report, HP Laboratories, June 2006.

[136] Gilman Tolle, Joseph Polastre, Robert Szewczyk, David Culler, Neil Turner, Kevin Tu, Stephen Burgess, Todd Dawson, Phil Buonadonna, David Gay, and Wei Hong. A macroscope in the redwoods. In *Proc. Int. Conf. Embedded Networked Sensor Systems*, pages 51–63, 2005.

[137] Gilman Tolle, Joseph Polastre, Robert Szewczyk, David Culler, Neil Turner, Kevin Tu, Stephen Burgess, Todd Dawson, Phil Buonadonna, David Gay, and Wei Hong. A macroscope in the redwoods. In *Proc. Int. Conf. Embedded Networked Sensor Systems*, pages 51–63, November 2005.

[138] B. Tremaine, P. A. Franaszek, J. T. Robinson, C. O. Schulz, T. B. Smith, M.E. Wazlowski, and P. M. Bland. IBM memory expansion technology. *IBM J. Research and Development*, 45(2):271–285, March 2001.

[139] Irina Chihaia Tuduce and Thomas Gross. Adaptive main memory compression. In *Proc. USENIX Conf.*, pages 237–250, April 2005.

[140] Robert A. van Engelen and Kyle A. Gallivan. An efficient algorithm for pointer-to-array access conversion for compiling and optimizing DSP applications. In *IWIA 01: Proceedings of the Innovative Architecture for Future Generation High-Performance Processors and Systems*, page 80, 2001.

[141] Tim Wark, Chris Crossman, Wen Hu, Ying Guo, Philip Valencia, Pavan Sikka, Peter Corke, Caroline Lee, John Henshall, Kishore Prayaga, Julian O'Grady, Matt Reed, and Andrew Fisher. The design and evaluation of a mobile sensor/actuator network for autonomous animal control. In *Proc. Int. Conf. Information Processing in Sensor Networks*, pages 206–215, April 2007.

[142] Geoffrey Werner-allen, Jeff Johnson, Mario Ruiz, Jonathan Lees, and Matt Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *European Wkshp. on Wireless Sensor Networks*, 2005.

[143] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis. The case for compressed caching in virtual memory systems. In *Proc. USENIX Conf.*, pages 101–116, April 1999.

[144] A. Wood, G. Virone, T. Doan, Q. Cao, L. Selavo, Y. Wu, L. Fang, Z. He, S. Lin, and J. Stankovic. ALARM-NET: Wireless sensor networks for assisted-living and residential monitoring. Technical report, University of Virginia, January 2006.

[145] Shunzo Yamashita, Takanori Shimura, Kiyoshi Aiki, Koji Ara, Yuji Ogata, Isamu Shimokawa, Takeshi Tanaka, Hiroyuki Kuriyama, Kazuyuki Shimada, and Kazuo Yano. A 15 $\times$ 15 mm, 1 μA, reliable sensor-net module: enabling application-specific nodes. In *Proc. Int. Conf. Information Processing in Sensor Networks*, pages 383–390, April 2006.

[146] Peng Yang, R.A. Freeman, and K.M. Lynch. Distributed cooperative active sensing using consensus filters. In *Proc. Int. Conf. Robotics & Automation*, pages 405–410, April 2007.

[147] Mohamed Younis and Kemal Akkaya. Strategies and techniques for node placement in wireless sensor networks: a survey. *Ad Hoc Networks*, 6(4):621–655, 2008.

[148] Jerry Zhao and Ramesh Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *Proc. Int. Conf. Embedded Networked Sensor Systems*, pages 1–13. ACM, November 2003.