ABSTRACT

Title of dissertation: STORAGE-CENTRIC WIRELESS SENSOR NETWORKS FOR SMART BUILDINGS

Baobing Wang, Doctor of Philosophy, 2013

Dissertation directed by: Professor John S. Baras
Department of Electrical and Computer Engineering

In the first part of the dissertation, we propose a model-based systems design framework, called WSNDesign, to facilitate the design and implementation of wireless sensor networks for Smart Buildings. We apply model-based systems engineering principles to enhance model reusability and collaboration among multiple engineering domains. Specifically, we describe a hierarchy of model libraries to model various behaviors and structures of sensor networks in the context of Smart Buildings, and introduce a system design flow to compose both continuous-time and event-triggered modules to develop applications with support for performance evaluation. WSNDesign can obtain early feedback and high-confidence evaluation of a design without requiring any intrusive and costly deployment. In addition, we develop a graphical tool that exposes a sequence of design choices to system designers, and provides instant feedback about the influence of a design decision on the complexity of system analysis. Our tool can facilitate comprehensive analysis and bring competitive advantage to the systems design workflow by reducing costly unanticipated behaviors.

One of the main challenges to design efficient sensor networks is to collect and process the data generated by various sensor motes in Smart Buildings efficiently. To make this task easier, we provide an abstraction for data collection and retrieval in the second part of the dissertation. Specifically, we design and implement a distributed database system, called HybridDB, for application development. HybridDB enables sensors to store large-scale datasets in situ on local NAND flash using a novel resource-aware data storage system, and can process typical queries in sensor networks extremely efficiently. In addition, HybridDB supports incremental ϵ-approximate querying that enables clients to retrieve a just-sufficient set of sensor data by issuing refinement and zoom-in sub-queries to search events and analyze sensor data efficiently. HybridDB can always return an approximate dataset with guaranteed maximum absolute (L_∞-norm) error bound, after applying temporal approximate locally on each sensor, and spatial approximate in the neighborhood on the proxy. Furthermore, HybridDB exploits an adaptive error distribution mechanism between temporal approximate and spatial approximate for trade-offs of energy consumption between sensors and the proxy, and response times between the current sub-query and the following sub-queries. The implementation of HybridDB in TinyOS 2.1 is transformed and imported to WSNDesign as a part of the model libraries.

STORAGE-CENTRIC WIRELESS SENSOR NETWORKS FOR SMART BUILDINGS

by

Baobing Wang

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2013

Acknowledgments

I owe my gratitude to all the people who have made this thesis possible and because of whom my graduate experience has been one that I will cherish forever.

First and foremost I would like to thank my advisor, Professor John S. Baras for giving me an invaluable opportunity to work on challenging and extremely interesting projects over the past four years. His deep insight on amazingly broad area of research, limitless energy and enthusiasm on challenging problems has been the most important support for my work. Dr. Shah-An Yang was also a great source of help and encouragement, to whom I am extremely grateful for the guidance and knowledge that he so generously shared. I would also like to thank Dr. Mark A. Austin, Dr. Shuvra S. Bhattacharyya, Dr. Gang Qu and Dr. Uzi Vishkin, for agreeing to serve on my thesis committee and for sparing their invaluable time reviewing the manuscript.

Sincerely thanks to Dr. Shanshan Zheng, Dr. Kiran Somasundaram, Kaustubh Jain, Dr. Ion Matei, Tuan Ta, Dr. Hua Chen, and other colleagues in the HyNet center and SEIL lab, who have enriched my graduate life in many ways and with whom I always had inspiring and fruitful discussions. Also I would like to thank Mrs. Kim Edwards for her great administrative support.

This dissertation is partially supported by DARPA and SRC through grant award 013641-001 of the FCRP, by the National Science Foundation (NSF) under grant award CNS-1035655, and by the National Institute of Standards and Technology (NIST) under grant award 70NANB11H148.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Wireless Sensor Networks and Smart Buildings

Buildings are some of the largest energy consumers in the world. Improving the energy efficiency of buildings is an important step towards a more sustainable lifestyle leading to significant cost savings for energy consumption, and a more comfortable environment for occupants. Due to their significant advantages, Wireless Sensor Networks (WSNs) will play a fundamental role in future Smart Buildings. It is possible to retrofit old buildings to enable sensing and monitoring with minimal changes, and even integrate WSNs with existing Building Energy and Control Systems (BECS) for finer grained distributed control. In addition, a large number of parameters can be measured to capture spatial and temporal distributions at a much finer granularity than other technologies available.

However, it is notoriously difficult to design efficient and reliable WSNs for Smart Buildings. Firstly, the design of such hybrid systems requires collaboration and optimization across multiple engineering domains. For example, some specialized teams are involved with the HVAC (Heating, Ventilation, and Air Conditioning) system design, while other experts focus on the protocol design and data collection in WSNs. The high coupling of various components requires that the various engineering teams collaborate and share their designs with each other regularly to ensure that the final design meets the overall design goals [2]. Secondly, the cyber-physical interactions must be thoroughly investigated. The success of separated tests of the BECS system and sensor network cannot guarantee that the overall system satisfies the requirements. Therefore, both subsystems must be tested in an integrated framework, which must enable various teams to design concurrently [3]. Thirdly, component reusability must be imposed. Since it is too costly to design such hybrid systems from scratch, the framework must be able to import existing component libraries. Finally, data collection and retrieval in WSNs is generally a tricky and tedious task, and the deployment of WSNs is complex due to the complicated interior structures of buildings [4].

The LoCal project[1] is very good example, which aims to design a network architecture for localized electrical energy reduction, generation and sharing. This project currently provides around 2000 distinct measurement channels that monitor electricity consumption, environmental quality data, HVAC parameters, weather data, etc [5]. In one of their experiments, 455 wireless power meters running TinyOS[2] over 6loWPAN/IPv6 are deployed in a commercial building for around one year, and over 900 million individual readings are collected (over 4000 readings per node per day on average) [6]. During their deployment, much effort was made to place the gateways to ensure load balancing, connectivity and short paths for the sensors to connect to the gateways.

[1]`http://local.cs.berkeley.edu/`

[2]`http://www.tinyos.net/`

1.2 Main Contributions

In this dissertation, our objective is to answer the following questions: (1) How to develop an integration framework for the design and evaluation of WSNs in Smart Buildings across multiple engineering domains; (2) How to store and retrieve the large amount of sensor readings efficiently to facilitate the access to various sensor readings for building applications. The solution to the first question can facilitate the system-level design, while the solution to the second question can facilitate the system implementation by abstracting data storage and transmission. The main contributions of this dissertation are summarized as follows.

1.2.1 Model-Based Systems Design Framework

To answer the first question, we propose a model-based systems design framework, called WSNDesign, which is SysML-centric with three key features. Firstly, WSNDesign provides a hierarchy of model libraries to model various behaviors and structures of WSNs in the context of Smart Buildings, including the models for applications, system services, computation and communication algorithms, physical platforms, and cyber systems (including environments and BECS systems). Event-triggered components are either modeled in SysML Statechart Diagrams, or imported from existing TinyOS protocol libraries. Continuous-time components are modeled in Simulink or Modelica and their behaviors are described by differential equations, which are then transformed to SysML and imported to WSNDesign. Therefore, with the help of WSNDesign, system engineers can take advantage of many existing TinyOS and Modelica libraries, rather than design every thing from scratch. In addition, it enables multiple design teams to work concurrently.

Secondly, WSNDesign can generate source codes and configuration scripts to evaluate the performance of the system by simulations. Although theoretical analysis produces immediate performance results, accuracy is often sacrificed to over-simplifications and assumptions, especially for large complex systems. With code generation from system models, WSNDesign can save system engineers the trouble of writing simulation codes manually. WSNDesign integrates the existing widely accepted simulators to increase the confidence of the simulation results.

Finally, WSNDesign provides an interactive tool to reduce the complexity of system analysis. Due to the interaction of connected components, exploration in existing formal verification tools typically makes the complexity of system analysis either high-order polynomial, or exponential in the system size. However, for a large class of systems, the essential complexity is linear in system size and exponential in treewidth, which means the previous notion of exponential complexity in system size is overestimated. WSNDesign reduce the complexity of system analysis using summary propagation on factor graphs transformed from SysML Parameter Diagrams [7], and expose a sequence of design choices to system designers to provide instant feedback about the influence of a design decision on the complexity of system analysis.

1.2.2 Data Storage and Incremental ϵ-Approximate Querying

The rest of this dissertation provides the answer for the second question. For many building applications, the data is needed only in aggregate form, such as the average power consumption over each hour or the peak power consumption for each day [1]. In addition, sometimes it is more interesting to derive usage patterns, rather than collect accurate individual readings. For example, based on the usage patterns derived from the readings of wireless energy plus-load meters, authors in [6] figured out which parts in the building and when they are wasting energy, which is very useful for improving the building management. Since data storage and querying are common desired features for such applications, it is preferable to provide these modules in the model libraries.

Existing centralized data acquisition techniques suffer from large energy consumption and traffic overhead, as all the readings are transmitted to the sink. In long-term deployments, it is preferable to store a large number of readings *in situ* and transmit a small subset only when requested [8, 9]. This framework becomes practically possible with the new generation NAND flash. Recent studies show that NAND flash is at least *two orders of magnitude cheaper* than communication and *comparable in cost* to computation [10]. Therefore, extending NAND flash to low-end sensor platforms can potentially improve in-network processing and energy-efficiency substantially.

However, due to the fundamentally different read and write semantics of NAND flash, and tightly constrained resource on sensor platforms, designing an efficient resource-aware data management system for flash-based sensor devices is a very challenging task. Existing techniques (e.g., [11–15]) are not applicable due to their large RAM footprints. Other works, such as TL-Tree [16] and FlashLogger [17], can only process simple time-based queries. More importantly, however, none of existing works take advantage of both the on-board *random-accessible* NOR flash that is quite suitable for index structures available in current sensor platforms, and external economical energy-efficient NAND flash with high-capacity, which is ideal for massive data storage.

For queries retrieving a relatively large set of readings, data approximate is a popular technique to reduce the traffic. Traditional methods [18–20] require users to specify fixed error bounds to address the trade-off between accuracy and overhead. However, in many scenarios, it is unfeasible and inefficient for users to determine in advance what error bounds can lead to acceptable results. On one hand, if an error bound is too tight, much energy will be wasted to retrieve more readings than needed, resulting in an over-qualified result. On the other hand, if the error bound is not tight enough, the set of readings returned by the query cannot produce a satisfactory result. In this case, the user needs to re-issue this query with a tighter error bound. Traditional schemes will treat it as an independent new query, and thus all readings that have been retrieved by the previous query will be transmitted again, resulting in much energy waste as well.

In the second part of the dissertation, we design and implement HybridDB, an efficient light-weight distributed database system for flash-based storage-centric WSNs. HybridDB exploits a novel resource-aware data storage system, called

HybridStore, to store and query sensor data *in situ* on each sensor mote. HybridStore has three key features. Firstly, it takes advantage of the on-board random-accessible NOR flash in current sensor platforms to guarantee that all NAND pages used by it are *fully occupied* and written in a *purely sequential* fashion, and expensive in-place updates and out-of-place writes to an existing NAND page are *completely avoided*. Thus, both raw NAND flash chips and FTL-equipped (Flash Translation Layer) flash packages can be supported efficiently. Secondly, HybridStore can process typical joint queries involving both time windows and key value ranges as filter conditions extremely efficiently, even on large-scale datasets. Finally, HybridStore can trivially support time-based data aging without any extra overhead, and provides an efficient failure recovery mechanism that guarantees the highest level of data consistency without the need for any checkpoint.

Based on HybridStore, HybridDB provides the support for incremental ϵ-approximate querying that enables clients to retrieve a just-sufficient set of readings by issuing sub-queries with decreasing error-bounds. HybridDB will return an approximate dataset with arbitrary maximum absolute (L_∞-norm) error bound, after applying temporal approximate locally on each sensor, and spatial approximate in the neighborhood on the proxy. In addition, HybridDB exploits an adaptive error distribution mechanism between temporal approximate and spatial approximate for trade-offs of energy consumption between sensors and the proxy, and response times between the current sub-query and following sub-queries. The implementation of HybridDB in TinyOS 2.1 can be transformed and imported to WSNDesign as a part of the model libraries. We also analyze its expected performance to integrate it into the theoretical performance estimation process of WSNDesign.

1.3 Thesis Organization

The rest of the dissertation is organized as follows. Part I focuses on WSNDesign, our model-based systems design framework. Chapter 2 describes the hierarchy of system model libraries and the integration of SysML and Simulink. Chapter 3 presents a modeling and co-simulation toolchain with the support for code generation and integration of TinyOS and Modelica. Chapter 4 discusses the interactive tool for reducing the complexity of system analysis.

Part II focuses on data storage and incremental ϵ-approximate querying. Chapter 5 discusses the background and formulates our problem. Chapter 6 describes HybridStore, an efficient flash-based data management system. Chapter 7 present HybridDB, an efficient distributed database system for incremental ϵ-approximate querying. Finally, Chapter 8 concludes this dissertation.

Part I

Model-Based Systems Design Framework

Chapter 2

WSNDesign: An Integrated Modeling and Simulation Framework

2.1 Introduction

Wireless Sensor Networks (WSNs) are engineered systems consisting of closely interacting physical environments, physical platforms, communication protocols and computation algorithms. The design of such hybrid systems requires a systems engineering view and an integrated design framework that can support joint event-triggered and continuous-time dynamics [21]. In addition, since hybrid systems usually are very complex and too costly to be designed from scratch, component reusability can never be overemphasized. Developing such a design framework for WSNs faces several challenges. For example, due to the wide variety of WSN applications and the heterogeneity of sensor platforms, it is difficult to figure out the primitive function modules, which are imperative for reusability. In addition, integrating the numerical solvers for continuous-time models with event-triggered models is an established, but far-from-trivial problem.

Several works have tried to improve the code reusability of sensor network protocols. Klues et al. [22] proposed a component-based architecture for the MAC layer in WSNs. Ee et al. [23] introduced a modular network layer to enable co-existing protocols to share and reduce code and resources consumed at run-time. However, both works focus on protocol implementations rather than system designs.

Viptos, a joint modeling and design framework for WSNs, was proposed in [24]. Mozumdar et al. [25] presented a similar work modeled in Simulink. They can analyze the performance of system designs by simulations and generate the TinyOS application codes. Another work [26], introduced a method to integrate Simulink and ns-2 for hybrid networked control systems. However, sensor behaviors in these works are tightly coupled with communication protocols, which makes their components hard to be reused.

Samper et al. [27] presented an approach for the formal modeling and analysis of WSNs at various abstraction levels. Formal model checking tools can be applied to verify their models of hybrid systems. However, trade-off analysis is not considered and the component reusability is not clearly supported in their approach. Moppet, a model-driven performance engineering framework for WSNs, was proposed in [28], which is the closest work to our framework. Moppet enables users to design WSN applications using the model libraries and estimate their performances using event calculus and network calculus without simulations. However, the continuous dynamic behaviors of physical environments are not considered.

In this chapter, we propose a model-based system design framework for WSNs, called WSNDesign, which applies system engineering principles to model both event-triggered and continuous-time components. Firstly, WSNDesign is proposed, which provides a hierarchy of system model libraries for applications, system services, computation and communication algorithms, physical platforms and physical environments to support a plug-and-play design fashion. Event-triggered components are modeled in SysML and statechart diagrams are exploited to model their behaviors. Continuous-time components are modeled in Simulink or Modelica and their behaviors are described by differential equations. To make our ideas more clear, the model libraries for the MAC layer, physical platforms, wireless channels and physical environments are briefly explained.

Secondly, based on WSNDesign, a system design flow is proposed, which can integrate both event-triggered and continuous-time modules. With the help of IBM Rational Rhapsody [29], Simulink and C/C++ source files can be generated automatically from WSNDesign, which can be used for performance study and interactive simulations.

Finally, a building thermal control system is used as the case study to demonstrate the reusability and flexibility of the proposed framework. In this example, we illustrate how hybrid systems can be easily developed using the modules in the model libraries, and how their performance can be studied. Using WSNDesign, developers can focus on the system design strategies, rather than implementation details that are usually not familiar to system experts.

The rest of this chapter is organized as follows. We introduce the proposed framework and design flow in Section 2.2. In Section 2.3, we describe the main modules in some model libraries. The case study is presented in Section 2.4. Finally, Section 2.5 summarizes this chapter.

2.2 Overview of the WSNDesign

In this section, we first introduce the proposed framework, with a hierarchy of model libraries to model various behaviors and structures of WSNs. Then we describe the design flow to develop applications with support for automatic code generation for simulations.

2.2.1 Hierarchy of System Models

We view a sensor network as an application-oriented data-centric service provider. Sensors collaborate to deliver services to accomplish the network missions, fulfilling its requirements and optimizing its performance subject to platform and environment constraints. The hierarchy of system model libraries in WSNDesign is shown in Fig. 2.1, aligned with their corresponding counterparts in the real world. WSNDesign provides the model libraries for applications, services, computation algorithms, communication protocols, physical platforms and cyber systems.

Application Model Library. A WSN application can be specified with function requirements, performance requirements, physical platforms and the physical environment where the sensor network will be deployed. The Application Model Library provides modules that can precisely describe common sensor network applications. In addition, special applications can also be modeled by extending proper models in the library.

Service Model Library. Most WSN applications share several common features that are used frequently, such as the query service to retrieve data locally

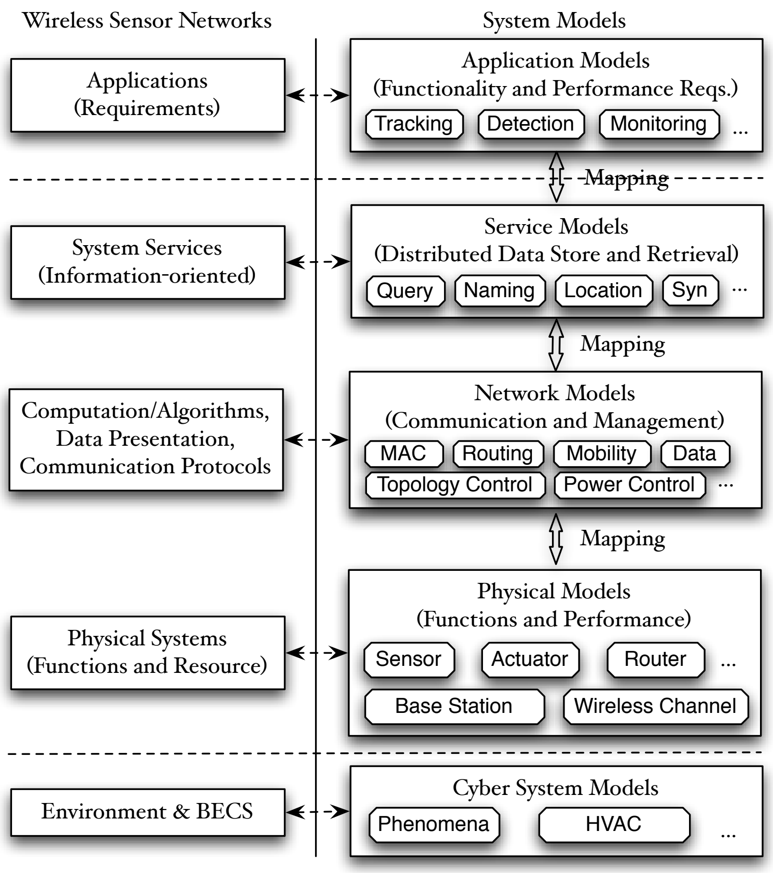

Wireless Sensor Networks
System Models
Applications (Requirements)
Application Models (Functionality and Performance Reqs.)
Tracking
Detection
Monitoring
...
Mapping
System Services (Information-oriented)
Service Models (Distributed Data Store and Retrieval)
Query
Naming
Location
Syn
...
Mapping
Computation/Algorithms, Data Presentation, Communication Protocols
Network Models (Communication and Management)
MAC
Routing
Mobility
Data
Topology Control
Power Control
...
Mapping
Physical Systems (Functions and Resource)
Physical Models (Functions and Performance)
Sensor
Actuator
Router
...
Base Station
Wireless Channel
Environment & BECS
Cyber System Models
Phenomena
HVAC
...

Figure 2.1: Hierarchy of System Models

or remotely, the naming service to uniquely identify motes locally or globally, the location service to compute the virtual or physical locations and regions of sensor motes, etc. The Service Model Library provides modules for these common services with interfaces to customize to fulfill the application requirements.

Network Model Library. This library resides in the center of WSNDesign, consisting of communication modules, computation modules and data management modules that are necessary to implement various algorithms and protocols to accomplish the upper layer services. By well defining their interfaces, different algorithms and protocols can be studied and compared systematically in a plug-and-play way, using components of the same functionalities and ports, but with different implementations.

Physical System Model Library. This library is composed of modules for various physical platforms in heterogeneous WSNs. Despite their different capacities and computation powers, these platforms can be viewed to be composed of at most four parts: CPU, sensor, transceiver and battery. We have distilled the various common primitives and parameters to describe these components and their ports. In addition, this library provides wireless channel models with different radio propagation models, channel fading models and bit error rates under different modulation schemes.

Environment Model Library. This library serves as the bridge between the continuous-time domain and the event-triggered domain. On one hand, physical environments usually exhibit continuous dynamic behaviors. On the other hand, algorithms and protocols in WSNs are usually event-driven and exhibit discrete dynamic behaviors. This library provides modules to exchange information between these two domains.

All event-triggered components are modeled in SysML using statecharts and primitive operations are implemented in C/C++. All continuous-time components are modeled using Simulink or Modelica, which are then compiled to generate S-Funtions and imported to SysML. Continuous data are passed through flow ports, while events and discrete data are exchanged via rapid ports.

2.2.2 System Design Flow

The objective of WSNDesign is to develop an integration framework for the design and evaluation of WSNs in Smart Buildings across multiple engineering domains. The integrated design environment and its associated design flow is shown in Fig. 2.2, which can compose both event-triggered and continuous-time modules, with support for model transformation and integration, complexity reduction of system analysis, and automatic code generation.

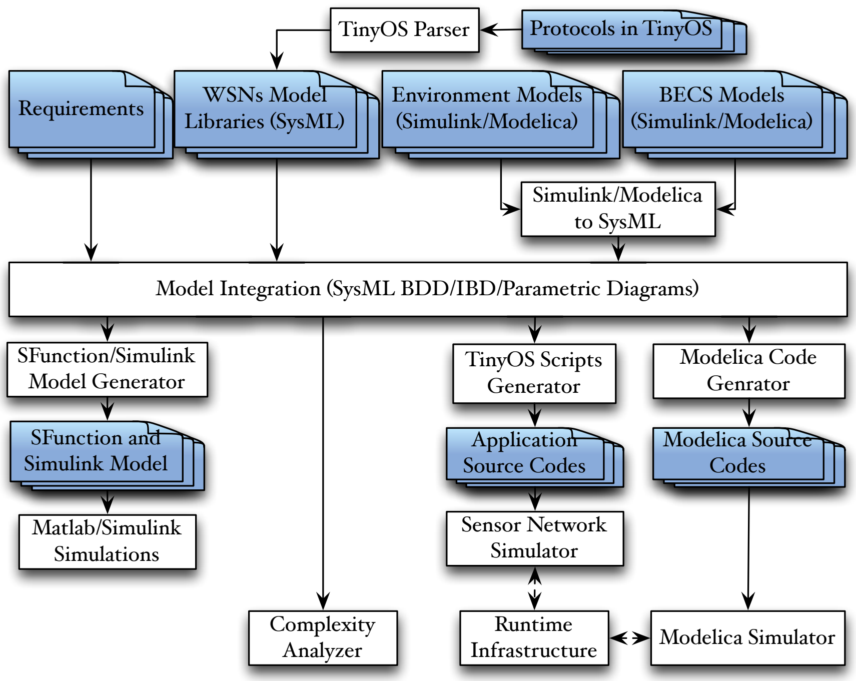

TinyOS Parser
Protocols in TinyOS
Requirements
WSNs Model Libraries (SysML)
Environment Models (Simulink/Modelica)
BECS Models (Simulink/Modelica)
Simulink/Modelica to SysML
Model Integration (SysML BDD/IBD/Parametric Diagrams)
SFunction/Simulink Model Generator
TinyOS Scripts Generator
Modelica Code Genrator
SFunction and Simulink Model
Application Source Codes
Modelica Source Codes
Matlab/Simulink Simulations
Sensor Network Simulator
Complexity Analyzer
Runtime Infrastructure
Modelica Simulator

Figure 2.2: Integrated Design Environment

System designs are developed by composing proper modules from the model libraries. System engineers can model event-triggered components using SysML Statechart Diagrams directly. However, since TinyOS is the most popular open-source operating system for low-power wireless devices, and many applications and protocols have been implemented in TinyOS, it makes no sense to model every thing from scratch. To reuse the existing TinyOS libraries, WSNDesign can transform their structures into the SysML format, and import them into the model libraries. WSNDesign refers to their TinyOS source codes for the behaviors of these modules, which will be used to generate simulation codes. Similarly, continuous-time components can be imported from existing Simulink or Modelica libraries, such as the Modelica Buildings Library[1].

After a system design is decided, system engineers must evaluate its performance in details. Although theoretical analysis produces instant performance results, accuracy is often sacrificed to oversimplifications and assumptions, especially for large complex systems. Simulation is probably the most popular method, which can predict the performance of complex and large networking systems that are theoretically very difficult to analyze without the need for expensive and time-consuming testbed experiments. WSNDesign can generate source codes and configuration scripts for simulations automatically from system models, which can save system engineers the trouble of writing simulation codes manually. In order to increase the confidence of the simulation results, WSNDesign integrates several existing widely accepted simulators for both Modelica and TinyOS.

Note that WSNDesign is independent from development tools, although some tools may be helpful to implement certain functions. In this dissertation, we choose IBM Rational Rhapsody[2] as the development environment, because it provides a

[1] `http://simulationresearch.lbl.gov/modelica`

[2] `http://www.ibm.com/software/products/us/en/ratirhapfami/`

complete open API to develop new plug-ins. Meanwhile, if all components are modeled in SysML and Simulink, it can generate C/C++ codes to simulate the system directly, or SFunction/Simulink models to simulate in Matlab. In this chapter, we adopt this method to take advantage of the functions provided by IBM Rhapsody. In Chapter 3, we develop a toolchain to integrate Modelica and TinyOS.

Another important process in systems design is investigate the correctness and optimality of the designed systems. However, as systems grow in size, the number of variables increase rapidly, which can make these problems computationally intractable. By partitioning a system into a graph of components, system engineers can understand the overall system by using a series of local analysis and a compositional technique to compose the results. However, most partitioning methods are ad-hoc and there are no quantitative metrics for measuring the complexity. Based on [7] that identified *treewidth* as a metric for the essential system complexity, WSNDesign provides an interactive tool to calculate the system treewidth, and expose a sequence of design choices to system engineers to provide instant feedback about the influence of a design decision on the complexity of system analysis. WSNDesign reduce the complexity of system analysis using summary propagation on factor graphs transformed from SysML Parameter Diagrams. Note that the objective of WSNDesign is not to formally verify or optimize the designed systems, but to help system engineers understand the implications of their design decisions on system complexity.

2.3 System Component Models

Modeling WSNs is generally a complex task due to the wide variety of WSN applications, the heterogeneity of physical platforms, and the complex interactions with their physical environments. In order to enhance the reusability, it is imperative to identify primitive function components in different layers and the interfaces for them to interact with each other. In this section, we briefly describe the main modules for physical platforms, the MAC layer, wireless channels and physical environments[3].

2.3.1 Modeling Wireless Sensor Networks

Sensor motes (wireless routers or base stations) consist of physical platforms and softwares (i.e., algorithms and protocols), communicating through wireless channels. Sensor motes provide ports to get the clear channel assessment (CCA), query environment phenomenon data, register themselves to the wireless channel component and send/receive packets.

2.3.1.1 Physical Platforms

A physical platform is composed of four parts: battery, CPU, sensor and transceiver. Their interactions in a typical composition of a physical sensor platform is shown in Fig. 2.4. We describe the transceiver module in detail as an example,

[3]For more details, please refer to `http://www.ece.umd.edu/~briankw/resources/Wang_WETICE_2012_Full.pdf`

whose behaviors are modeled as a state machine in Fig. 2.3.

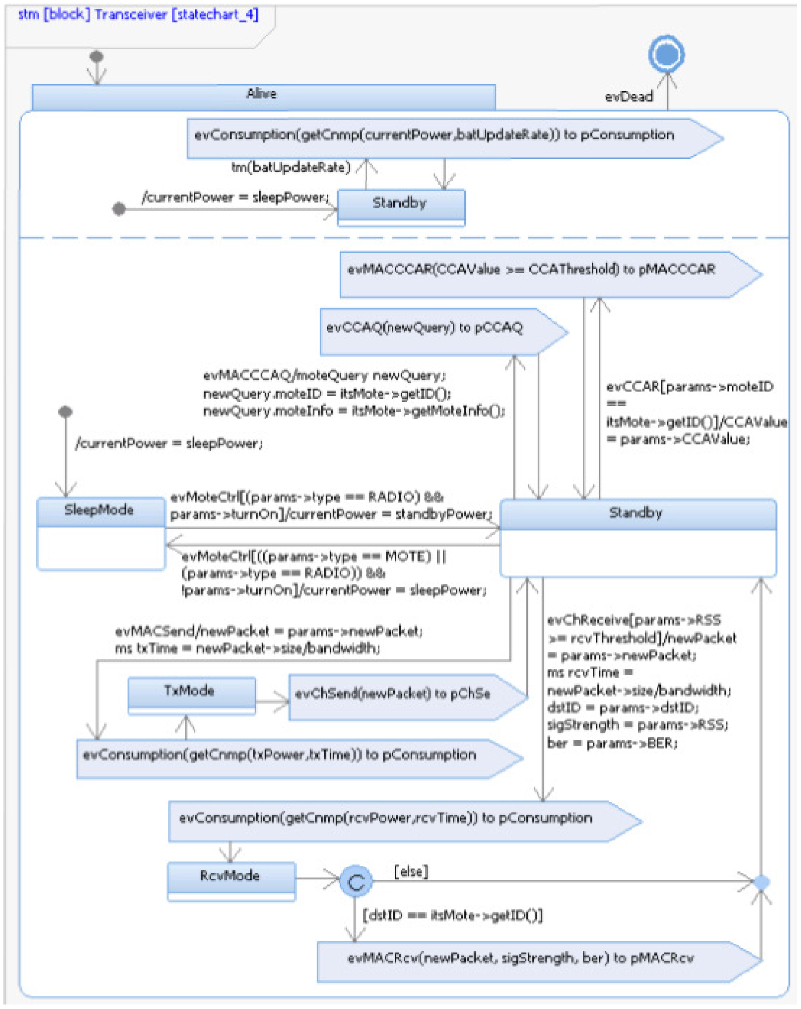

stm [block] Transceiver [statechart_4]
Alive
evDead
evConsumption(getCnmp(currentPower,batUpdateRate)) to pConsumption
tm(batUpdateRate)
/currentPower = sleepPower;
Standby
evMACCCAR(CCAValue >= CCAThreshold) to pMACCCAR
evCCAQ(newQuery) to pCCAQ
evMACCCAQ/moteQuery newQuery;
newQuery.moteID = itsMote->getID();
newQuery.moteInfo = itsMote->getMoteInfo();
evCCAR[params->moteID
==
itsMote->getID()]/CCAValue
= params->CCAValue;
/currentPower = sleepPower;
SleepMode
evMoteCtrl[(params->type == RADIO) &&
params->turnOn]/currentPower = standbyPower;
Standby
evMoteCtrl[((params->type == MOTE) ||
(params->type == RADIO)) &&
!params->turnOn]/currentPower = sleepPower;
evMACSend/newPacket = params->newPacket;
ms txTime = newPacket->size/bandwidth;
evChReceive[params->RSS
>= rcvThreshold]/newPacket
= params->newPacket;
ms rcvTime =
newPacket->size/bandwidth;
dstID = params->dstID;
sigStrength = params->RSS;
ber = params->BER;
TxMode
evChSend(newPacket) to pChSe
evConsumption(getCnmp(txPower,txTime)) to pConsumption
evConsumption(getCnmp(rcvPower,rcvTime)) to pConsumption
RcvMode
C
[else]
[dstID == itsMote->getID()]
evMACRcv(newPacket, sigStrength, ber) to pMACRcv

Figure 2.3: Behavior Model of a Transceiver Using Statechart

Each transceiver has four power states, whose levels are given by `sleepPower`, `standbyPower`, `txPower` and `rcvPower`, respectively. To support the Unit Disk Graph (UDG) model, the `txRange` attribute can specify the transmission range. The transceiver sends the energy consumption information to the battery periodically and transits to the termination state if an `evDead` event is received. In addition, it will transit its power state accordingly if an `evMoteCtrl` message is received.

If an `evMACCCAQ` query is received from the MAC layer, which requires to poll the channel to get the CCA information, the transceiver will forward this query with the mote ID to the wireless channel component via `pCCAQ`. The reply `evCCAR` that contains the strength of the strongest signal in the channel near this mote is returned by the wireless channel component through `pCCAR`. The transceiver compares this signal strength with its carrier sense threshold (specified by the `CCAThreshold` attribute) and sends the result (`true` or `false`) to the MAC layer through `pMACCCAR`.

If an `evMACSend` request that contains the packet to be sent is received from the MAC layer, the transceiver will forward this packet to the wireless channel component through `pChSend`. Furthermore, the amount of energy consumed to send this packet is sent to the battery component. If an `evChReceive` event that contains the packet forwarded by the wireless channel component is received from `pChReceive`, the transceiver will check the received signal strength (RSS) and the destination of the packet. If it is the destination and the RSS exceeds its receive sensitivity (specified by `rcvThreshold`), it will forward this packet to the MAC layer through `pMACRcv`. Otherwise, this packet will be dropped. Similarly, the energy consumption for packet processing is sent to the battery component.

2.3.1.2 MAC Layer Components

A MAC layer component provides the ports for upper layers to send and receive packets, set the transmission power of the transceiver and control the power states of the physical sensor platform. The main subcomponents in this layer are described as follows, some of which are developed based on [22]. A composition example is shown in Fig. 2.5.

The *Low Power Listener (LPL)* component adjusts the transceiver's power state based on channel activity. Both the fixed LPL listener and the periodic LPL listener are provided in WSNDesign. The *CSMA/CA Channel Access* component is responsible to gain the channel access right for a transmission using the CSMA/CA mechanism. The *CSMA/CA Sender* component is responsible to send a packet using the CSMA/CA mechanism. The *Slot Manager* component manages the slot schedule for TDMA mechanism. The *TDMA Sender* component is similar to the CSMA/CA Sender, except that packets are sent using the TDMA mechanism. The *Receiver* component is responsible to broadcast received packets to the MAC Controller and other protocol-specific components for further process. The *Queue Manager* component is responsible to buffer packets in the MAC layer.

The *MAC Controller* component is the only one that needs to be customized by users. This component specifies the control logic of a MAC protocol. Every MAC protocol should extend this abstract component and implement the protocol-specific behaviors. The ports to interact with other modules have been defined, including components in upper layers and other components in the MAC layer.

2.3.1.3 Wireless Channels

Wireless channel components model various wireless channels with different radio propagation models, channel fading models and bit error rates (BERs) under different modulation schemes. Each network instance usually has only one wireless channel component, to which all sensor motes, actuators, wireless routers and base stations must register themselves with their IDs, physical positions and transmission powers/ranges.

Since the channel component interacts with all nodes in a network instance, it may not be able to process all requests immediately. Therefore, the channel component needs to buffer the received requests in FIFO queues. In addition, the channel component needs to maintain the information of all nodes in the network, including their IDs, physical positions and transmission powers/ranges. Furthermore, this component needs to maintain the information of all ongoing transmissions in the channel, including the physical positions and transmission powers/ranges of the senders, the start time-stamps and their required transmission times. These information are essential for the evaluation of CCAs and BERs.

2.3.2 Modeling Physical Environments

Environment phenomena (e.g., temperature and humidity) usually exhibit continuous dynamic behaviors, which are typically described by differential equations according to their physical laws and modeled using Matlab/Simulink or Modelica. Simulink is a generic data-flow simulation tool good for modeling control systems.

Simulink models are first built using the Embedded Coder in Matlab to generate C/C++ source codes, and then imported as SysML blocks to Rational Rhapsody. Modelica is a more powerful topological-based modeling language with support for symbolic manipulation of equations and non-causality, which makes it excellent for modeling plants and physical world. Modelica models should be transformed to Simulink S-Functions first [30]. The outputs of their solvers are forwarded to the *Phenomenon* module through flow ports.

The *Phenomenon* component serves as the interface between the continuous-time domain and the event-triggered domain. The basic Phenomenon component periodically sends new phenomenon information to the Environment component. The *Environment* component models the propagation of information that are received from the Phenomenon module. It accepts queries from sensors, computes the phenomenon values based on the propagation model and the distances between sensors and the phenomenon, and returns the results to sensors. Several Environment modules can be composed in a network for different environment regions. For example, two such modules are included in our case study with one for each room.

2.4 Case Study

WSNDesign proposed in this chapter is intended to provide a reusable and extensible mechanism for system design and performance simulations. In this section, we present a simple building thermal control system as the case study to demonstrate the composability, reusability and power of WSNDesign.

2.4.1 Building Thermal Control System

A building thermal control system is responsible to control the temperature inside a building so that people can feel comfortable and equipments can work in a optimal condition. In addition, it also needs to reduce the energy consumed by heaters and air conditioners (ACs).

In this case study, we consider a simple building that consists of two large rooms: the living room and the data center. Each room has a desired temperature, which is usually 22 °C and much higher than the environment temperature in the winter. Therefore, a heater is needed in the living room to generate warmth. On the other side, the temperature in the data center will naturally rise because the large amount of electrical power used by the computer systems will heat the air. Consequently, an AC is needed in the data center to keep the temperature at the desired level.

An efficient way to reduce the energy consumption is to use the heat emitted by the computer systems to heat the living room through a pipe. A central control system decides when the heater, AC and pipe should be turned on or off. One temperature sensor is deployed in each room, which sends the room temperature to the control system in the base station through the wireless channel. The commands from the control system are sent to the heater, AC and pipe directly.

In this case study, we assumed the IEEE 802.15.4 unslotted CSMA/CA [31] is used as the MAC protocol, and both the two sensors and base station can communicate with the personal area network (PAN) coordinator directly. The compositions of the main components are introduced as follows.

2.4.1.1 Physical Platforms

The temperature sensors, base station and PAN coordinator can be composed using components from the physical system model library. The internal composition of a temperature sensor is shown in Fig. 2.4. The base station and PAN coordinator can be composed in the similar way but without the sensor component.

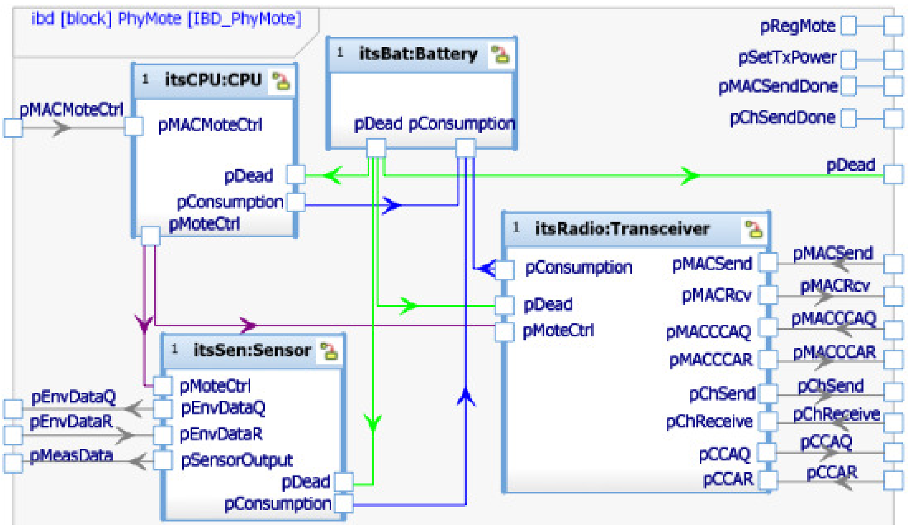


Figure 2.4: SysML Internal Block Diagram for Physical Sensor Motes

2.4.1.2 IEEE 802.15.4 MAC Protocol

The unslotted CSMA/CA MAC protocol for each node can be composed using components from the network model library. The internal composition of the MAC protocol for the PAN coordinator is shown in Fig. 2.5. Both the sensors and base station act as RFDs, whose MAC protocols can be composed in the similar way but without the queue manager component, and the PAN controller component is replaced with a RFD controller component.

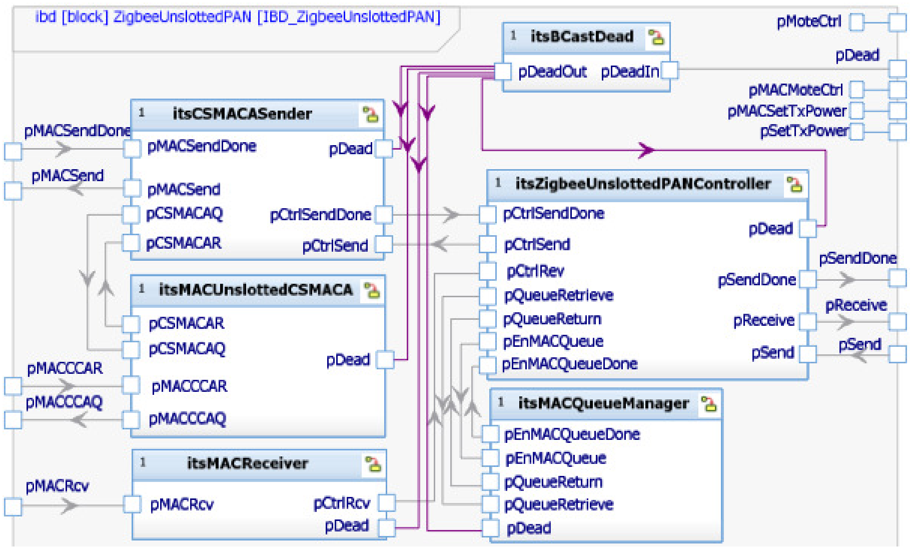


Figure 2.5: SysML Internal Block Diagram for the IEEE 802.15.4 Unslotted CSMA/CA Mode for the PAN Coordinator

2.4.1.3 Building Thermal Model in Simulink

The control system and building thermal dynamics are modeled in Simulink. The thermal dynamics model of the Data Center is shown in Fig. 2.6 as an example.

The control system decides when the heater, AC and pipe should be turned on or off based on the following rules:

- Heater: turn on if $T_{LR} \leq 20$ and turn off if $T_{LR} \geq 24$
- AC: turn on if $T_{DC} \geq 24$ and turn off if $T_{DC} \leq 20$
- Pipe: turn on if $T_{LR} \leq 22$ and turn off if $T_{LR} \geq 24$

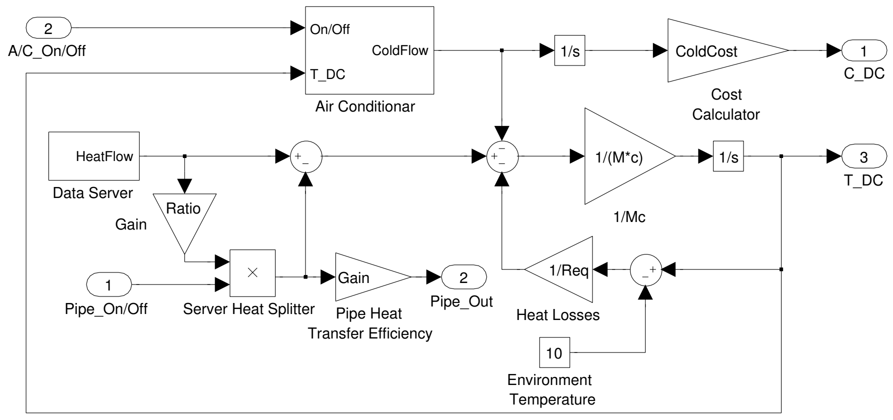


Figure 2.6: Simulink Thermal Dynamics Model of the Data Center

where T_{LR} and T_{DC} are temperatures of the living room and data center, respectively. Here, a variance of 2 °C around the desired temperature is allowed to avoid oscillations.

2.4.1.4 Overall System Composition

After all required components have been composed using the model libraries, they can be connected together to model the whole system. The overall system composition of our case study is shown in Fig. 2.7. The channel component with the ITU indoor propagation model [32], Rayleigh fading and BPSK modulation scheme is selected for this system. The connections between the channel component and the node components are not shown here for clarity. Ports with the same name (indicated by the same color) should be connected. The `SL_ControlSystem` component is used to import the Simulink models by applying the `<<SimulinkBlock>>` stereotype.

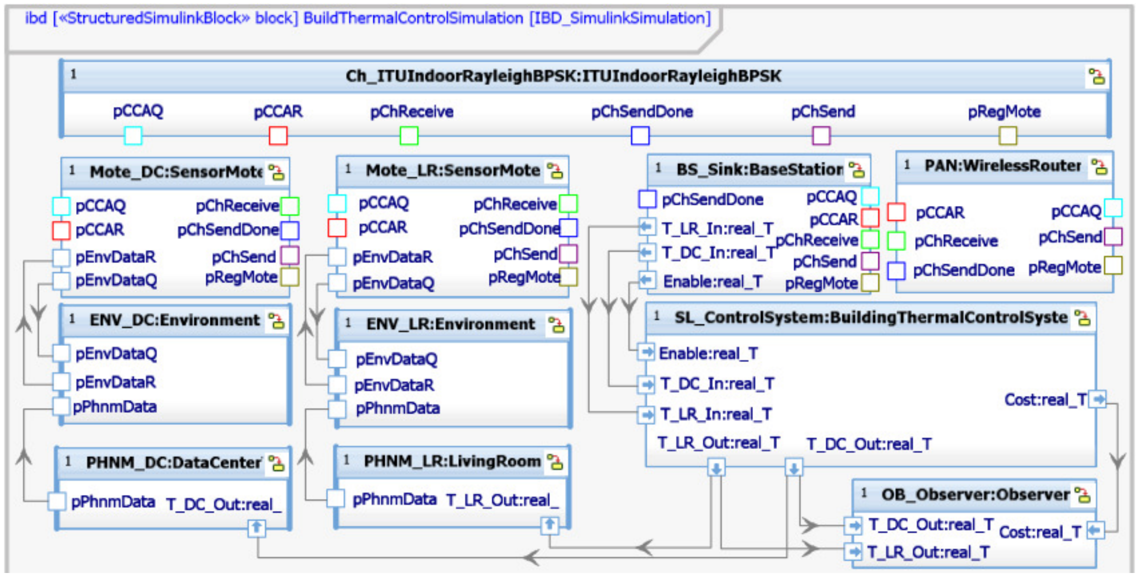


Figure 2.7: SysML Internal Block Diagram for the Whole System

2.4.2 Evaluation

The Simulink source file for the overall system is generated from the Simulink structure block in Figure 2.7 automatically, which is then simulated in Matlab/Simulink. The following four scenarios are considered in our simulations:

- **Wireless + No Pipe**. The temperatures are collected to the base station using the WSN, but the pipe is never turned on. Each sensor measures once every 5 seconds.

- **Wireless + Pipe** (5s). Similar to the above scenario, but the pipe feature is enabled.

- **Wireless + Pipe** (60s). Similar to the above scenario, but the sensor sleep interval is 60 seconds.

- **Wired + Pipe**. The room temperatures are fed back to the control center directly. This scenario is used as the reference to study the impacts of the delays in the WSN.

The temperatures of the environment, the heater, the AC and the air flow from the computer systems are 10 °C, 50 °C, 4 °C and 50 °C, respectively. The air flow rate of the AC, the heater and the computer systems are 2 kg/s, 2 kg/s and 0.5 kg/s, respectively. We assume 50% of the air flow from the computer systems can be piped out and 30% of their heat can be delivered to the living room. The MAC protocol uses the default parameter values specified in [31].

The initial temperature of both rooms is 20 °C. The simulation results for the first 2 hours of the room temperatures and cost are shown in Figure 2.8, and the working statuses of the AC, heater and pipe are shown in Figure 2.9. The results indicate that the pipe is working efficiently, which can decrease the working time of the heater and AC, and thus reduce the total electricity cost by 24%. When the sensors wake up to measure the temperature once every 5 seconds, the impacts of the delays on the system performance are negligible. However, if the interval between two successive measurements is increased to 60 seconds, the room temperatures may cross the desired boundaries, which should be avoided. This can be used to study the trade-off between the system performance and energy efficiency of the sensor motes.

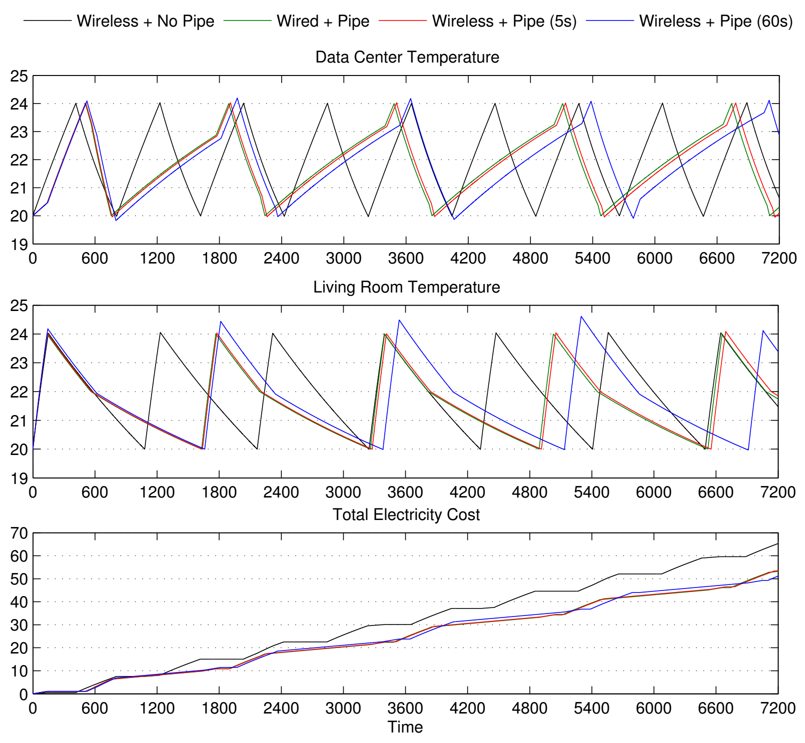


Figure 2.8: Room Temperatures and Total Electricity Cost

2.5 Summary

In this chapter, we have proposed a model-based system design framework for WSNs, which can model both continuous-time and event-driven components, and integrate them by composition for performance study by simulations. SysML, Simulink and Modelica that are standard modeling languages in the industry are used to develop the model libraries. The main component models for physical platforms, the MAC layer, wireless channels and physical environments are described briefly. A hybrid system is used as the case study to demonstrate the composability, reusability and flexibility of WSNDesign.

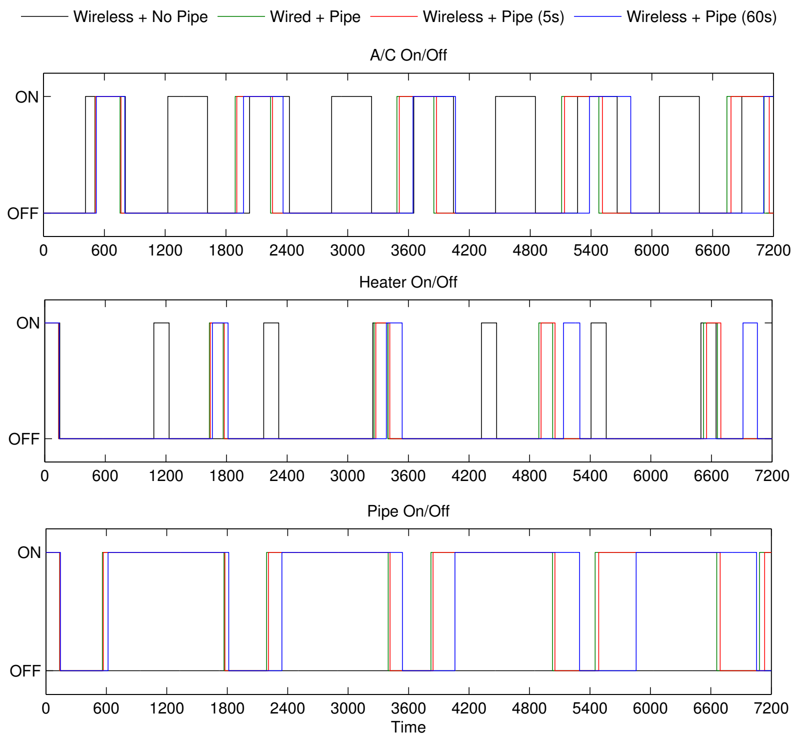


Figure 2.9: Working Statuses of the A/C, Heater and Pipe

Chapter 3

HybridSim: A Modeling and Co-simulation Toolchain

3.1 Introduction

One of the most pervasive applications of Cyber-Physical systems (CPS) is to integrate cyber environment and physical systems through computing and networking [26]. Traditionally, sensor measurements are fed to controllers through wired networks, which are also used to deliver control commands to actuators. However, for some applications (e.g., Smart Buildings, manufacture monitoring, etc.), it can be very expensive and complex to deploy and evolve such wired networks. For some other applications (e.g., unmanned aerial vehicles, healthcare monitoring, etc.), it will be impossible to use wired networks for communications. Therefore, it is preferable to integrate computational and physical devices through self-organized wireless networks.

However, wireless networks suffer from unreliable wireless channels, especially in indoor applications and body-area networks. It is imperative to understand how wireless networks and control systems affect each other as a function of network traffic, topology and background interference, in terms of the stability and performance of control systems before deployment [33]. Considering the complexity of CPS, simulations are preferable to pure mathematical analysis. Given the multi-domain nature of CPS, it is more appropriate to use a heterogeneous simulation environment to study system dynamics. However, the integration of numerical solvers for continues-time models and simulators for event-triggered models is an established, but far-from-trivial problem [34].

In addition, different domain engineering groups generally exploit different domain languages and tools to model and evaluate their designs, which may not be familiar to systems engineers. It is error-prone and inefficient if systems engineers have to work with these domain languages and tools directly to design and evaluate overall systems. Instead, the outputs of domain groups should be integrated into a unified framework that is friendly to systems engineers [21].

In this chapter, we design and implement an integrated modeling and co-simulation toolchain, called HybridSim, for the design and evaluation of CPS. HybridSim has three key features. Firstly, HybridSim provides a unified framework for systems engineers to integrate the outputs of domain groups to design overall systems. Specifically, we consider that a CPS consists of three main subsystems: 1) a cyber environment in which the CPS is deployed; 2) a wireless sensor and actuator network that is used to deliver sensor measurements and control commands; and 3) a control and computation system that is responsible to process information and make decisions. We assume the second subsystem is designed and implemented in TinyOS that is the most popular operating system for low-power wireless devices, and the other two subsystems are modeled and simulated in Modelica. HybridSim can transform and import TinyOS implementations and Modelica models into SysML blocks. Section 3.6 discusses how to extend HybridSim to support other domain tools and languages.

Secondly, systems engineers can design CPS by using the imported SysML blocks in a drag-and-drop fashion. Such SysML blocks appear as black boxes and all domain-specific details are hidden from systems engineers. HybridSim provides a set of SysML stereotypes for systems engineers to configure overall systems and provide guidance for code generation and co-simulations.

Finally, HybridSim can generate configuration scripts and simulation modules for domain implementations and models automatically from SysML designs. Based on the Functional Mock-up Interface (FMI) standard [35], HybridSim can co-simulate these subsystems by leveraging the advantages of their respective professional simulators and emulators. HybridSim provides a robust and flexible mechanism for data exchange and synchronization between these subsystems.

The convenience and efficiency of HybridSim is demonstrated by using a comprehensive hydronic heating system model for Smart Buildings as the case study. We investigate the impact of sampling rates, background traffic, and network sizes in wireless sensor networks on the performance of the control system modeled in Modelica.

The rest of this chapter is organized as follows. We discuss related work in Section 3.2 and introduce some background briefly in Section 3.3. In Section 3.4, we explain the design and implementation of HybridSim in details. The case study is presented in Section 3.5. Finally we discuss some observations, limits and future work for HybridSim in Section 3.6 and conclude this chapter in Section 3.7.

3.2 Related Work

Since the design and synthesis of CPS has become a hot topic in both industry and academia, several co-simulation paradigms have been proposed. TrueTime [36] extends Matlab/Simulink with libraries for co-simulation of controller task execution in real-time kernels, network transmissions, and continuous plant dynamics. However, physical and MAC layer protocols must be modeled from scratch, which is complex and time-consuming. In addition, providing support for higher layer protocols in TrueTime can be a formidable task because they generally utilize complex algorithms that are distributed in nature and encompass multi-hop communications [33].

Viptos [24] extends TOSSIM [37] by providing interrupt-level simulation of actual TinyOS programs, with packet-level simulation of the network, while allowing developers to use Ptolemy II to model the physical environment and other parts of the system. However, information can only flow from Ptolemy II to TOSSIM because there is no way for Ptolemy II to receive data from TOSSIM. Therefore, Viptos cannot be used to investigate the impact of network dynamics on the performance of overall CPS.

The Modelica/ns-2 co-simulation platform [33] integrates Modelica and ns-2 for CPS, with ns-2 deciding their communication times. Therefore, sending data between Modelica and ns-2 in response to events generated inside Modelica is not supported. The synchronization mechanism is improved later in [38]. NCSWT (Networked Control System Wind Tunnel) [26] integrates Matlab/Simulink with ns-2 according to the High Level Architecture (HLA) standard. To conform to the HLA standard, the time management mechanisms of both simulators are modified to enable synchronization between them. However, the overhead of NCSWT is very large, which takes more than one hour to complete a 98-second simulation. There are also other similar works on the integration of Matlab/Simulink and ns-2 [39,40].

In all the above work, simulators involved in co-simulations are tightly coupled with each other, and even need to be modified to enable data exchange and synchronization. Comparatively, HybridSim adopts the FMI standard, and only interacts with FMI-compatible API that are supported by more than 40 commercial and open-source tools[1]. This implies the extensibility of HybridSim to integrate other tools. In addition, existing works require systems engineers to work with domain languages and tools directly, while HybridSim enables systems engineers to work with only SysML blocks by transforming and importing domain implementations and models.

3.3 Background

In this section, we briefly introduce the tools and standards based on which HybridSim is designed and implemented.

[1]Full tool list: `https://www.fmi-standard.org/tools`

3.3.1 Functional Mock-up Interface

FMI [35] is a tool independent standard to support both model exchange and co-simulation of dynamic models using a combination of XML files and compiled C-code. FMI provides a set of standard functions to exchange data between subsystems and synchronize them in communication steps. These subsystems are called FMI slaves, while the co-simulation coordinator is called FMI master. FMI compliant models are referred to as Functional Mock-up Units (FMUs)

3.3.2 TinyOS and Avrora

TinyOS [41] is an open source operating system (OS) designed for low-power wireless devices, such as those used in sensor networks, ubiquitous computing, personal area networks, smart buildings, and smart meters. TinyOS emphasizes reacting to external events, extremely low-power operation and small memory footprint. Rather than a monolithic OS, TinyOS is object-oriented, providing a set of components that are included as-needed in applications by extending the C language with concepts of interfaces, modules and configurations. TinyOS is the most popular OS that is widely used to design and implement wireless sensor networks.

Avrora [42] is a set of emulation and analysis tools for programs written for MicaZ and mica2 sensor platforms. Avrora contains a flexible framework for emulating and analyzing executable binary files compiled from TinyOS applications, providing a clean Java API and infrastructure for experimentation, profiling, and analysis. Unlike TOSSIM [37], Avrora provides cycle-accurate instruction-level timing accuracy.

3.3.3 Modelica

Modelica [43] is an object-oriented language for modeling of large, complex, and heterogeneous systems. Models in Modelica are mathematically described by differential, algebraic and discrete equations. Modelica is designed so that available, specialized algorithms can be utilized to enable efficient handling of large models having more than one hundred thousand equations. Dymola [30] is a commercial modeling and simulation environment based on Modelica, which can compile Modelica codes, solve equations and simulate Modelica models. Specifically, Dymola can generate an FMU by wrapping a whole Modelica system model.

3.4 HybridSim: Design and Implementation

In this section, we first introduce the workflow of HybridSim, and then describe the model integration environment and co-simulation toolchain in more details.

3.4.1 HybridSim Workflow

HybridSim is an important module of WSNDesign. Its workflow is shown in Fig. 3.1, which roughly consists of three major steps: model transformation, system configuration, and co-simulation.

Model Transformation. The design of a cyber-physical system usually involves multiple engineering groups from different domains. For example, to design a

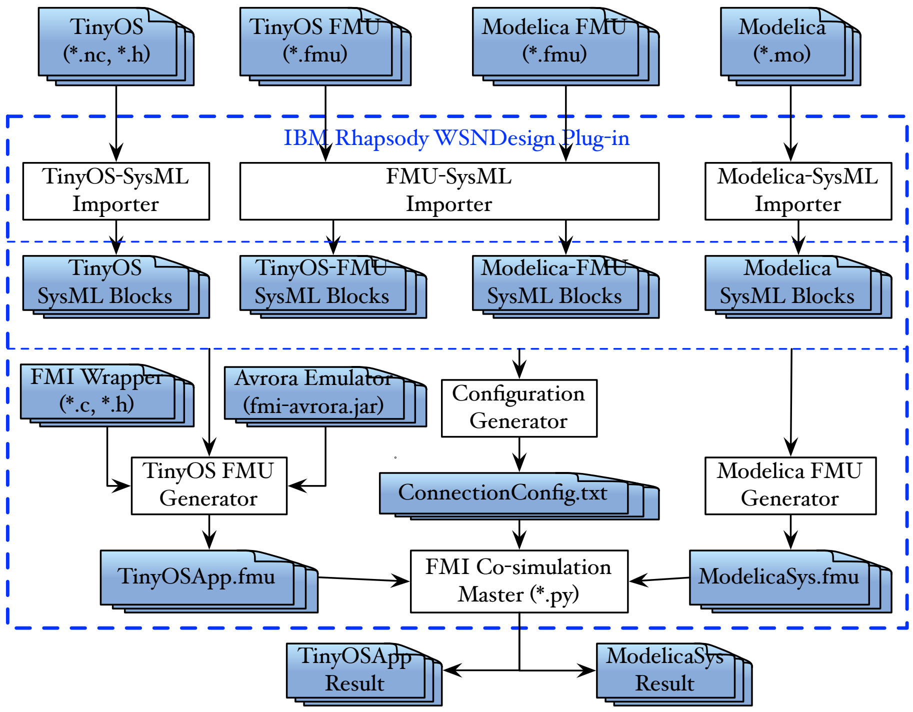


Figure 3.1: HybridSim Toolchain Workflow

smart building, we can have one group working on the HVAC (heating, ventilation, and air conditioning) system, and another group working on the communication network. Generally, different domain groups exploit different languages and tools to model and evaluate their designs, which may not be familiar to systems engineers. HybridSim solves this problem by transforming and importing domain models into SysML, which is widely used for model-based systems engineering. Currently HybridSim can transform TinyOS components, Modelica modules and FMUs, and import them as SysML blocks.

System Configuration. Systems engineers can create SysML Block Definition Diagrams and SysML Internal Block Diagrams, by selecting TinyOS SysML blocks, Modelica SysML blocks and FMU SysML blocks to set up a system-level simulation environment to evaluate the performance and dynamics of an overall system. Particularly, systems engineers need to configure simulation parameters for both TinyOS emulator and Modelica simulator, and specify information exchanging between them.

Co-simulation. After systems engineers have set up a simulation environment, HybridSim will compile TinyOS application codes and Modelica models, and generate their corresponding FMUs. Meanwhile, related configuration files and simulation scripts are generated automatically as well. Then HybridSim launches the FMI co-simulation master to co-simulate TinyOS FMU and Modelica FMU, and outputs their corresponding results.

3.4.2 SysML-centric Integration Environment

HybridSim is developed according to the following principles:

- Domain engineers work with the languages and tools with which they are familiar. They should not be required to re-implement their designs just for overall system simulation and evaluation. Ideally, domain engineers just need to provide standard interfaces for systems engineers to exchange information and synchronize with other domain simulators.
- Systems engineers only need to work with their favorite languages and tools as well. For overall system integration and simulation, they take the outputs of domain engineering groups as black boxes. Systems engineers only need to work with the interfaces provided by domain implementations and models.

- Systems engineers should take advantage of exiting professional simulators/emulators in each domain for overall system evaluation. In addition, the burden to set up a co-simulation environment should be minimized for systems engineers.

Support Modelica. As SysML is widely used for model-based systems engineering, HybridSim selects SysML as the working language for systems engineers. Meanwhile, Modelica is becoming increasingly popular to model CPS in both industry and academia in recent years. For example, researchers at the Lawrence Berkeley National Laboratory have developed a Modelica Buildings Library with dynamic simulation models for building energy and control systems in Smart Buildings [44]. Therefore, HybridSim supports Modelica as a very important domain language. Specifically, HybridSim can import Modelica models in two formats into SysML blocks: both Modelica FMUs and Modelica source codes.

The FMU-SysML Importer reads the model description (i.e., *modelDescription.xml*) of a Modelica FMU to figure out its input and output ports, and then creates a SysML block with corresponding input and output flow ports. In addition, the `ModelicaFMU` stereotype is applied to this SysML block, which has a tag referring to the original FMU for simulation code generation.

Similarly, the Modelica-SysML Importer parses Modelica source codes to extract its input and output ports information, and then generates SysML blocks. Unlike SyM [45], HybridSim only extracts the high-level structure information of a Modelica module, and ignores its internal component connections and behavior definitions. Therefore, this importer is much simpler and very stable. The `ModelicaSource` stereotype is applied to generated SysML blocks to refer their behaviors to the original source codes, and specify which tool should be used to generate FMUs from these SysML blocks. Currently, Dymola is the only tool supported by HybridSim. However, it is very easy to extend HybridSim to support OpenModelica or other Modelica tools.

Support TinyOS. WSNs will play a fundamental role in future CPS, such as Smart Buildings [46]. As TinyOS is the most popular operating system to develop sensor applications, HybridSim supports TinyOS and its programming language, nesC, for communication networks design. Similarly, HybridSim can transform and import TinyOS components in two formats into SysML blocks: TinyOS FMUs and nesC source codes.

As Modelica FMUs, TinyOS FMUs are transformed and imported by the FMU-SysML Importer as well. However, the `TinyOSFMU` stereotype is applied to their corresponding SysML blocks, which provides additional tags for co-simulation setup in addition to the reference to the original FMU. To the best of our knowledge, HybridSim is currently the only tool that can generate TinyOS FMUs. Considering that design groups of communication systems may not be familiar with SysML and IBM Rational Rhapsody, an independent version of TinyOS FMU Generator of HybridSim is also available.

To import TinyOS components from nesC source codes, HybridSim takes advantage of the XML output feature of the nesC compiler, from where input and output interfaces of TinyOS components are extracted. If a component does not have any interface (i.e., it is a complete TinyOS application), the `TinyOSSource` stereotype is applied to its corresponding SysML block. Otherwise, `TinyOSComponent` is applied.

Systems engineers can integrate a sensor network implementation into the overall system in two ways from source codes. The easiest way is to select an existing SysML block tagged with `TinyOSSource`. The other way is to create a new SysML block that is applied with `TinyOSSource`, and then specify a reference to an executable binary file, or create an Internal Block Diagram for it. Systems engineers can select and connect existing TinyOS components in a drag-and-drop fashion in this diagram to specify its source code. This enables systems engineers to develop simple sensor network applications for partial evaluation of overall systems before real applications are finished. Note that HybridSim only has a limited support for the second method, without interface compatibility checking for connections. In addition, although HybridSim can make use of existing imported TinyOS components, systems engineers cannot modify their source codes or create new TinyOS components in HybridSim. This is reasonable because systems engineers are not supposed to directly work with domain languages.

In order to exchange information with Modelica models, HybridSim provides a simple mechanism that allows systems engineers to input data coming from Modelica simulators into sensors, and vice verse. An example is shown in Fig. 3.3. To input data to sensors on a mote, systems engineers just need to add input flow ports to the SysML block (tagged with `TinyOSSource`) representing the application to be run on that mote. The name of an input port must exactly be the type of the target sensor (e.g., temperature, light, etc.). Currently, only scalar-valued sensors are supported. To get data out of motes, output flow ports should be added to their corresponding SysML blocks. The name of an output port must exactly be the name of a variable declared with module-scope in some TinyOS module. The `TinyOSVariable` stereotype should be applied to that port to specify in which module that variable is declared.

3.4.3 FMI-based Co-simulation Toolchain

HybridSim adopts the FMI standard to co-simulate TinyOS and Modelica, and synchronizes and exchanges information between their respective simulators by calling their FMI-compatible API. The implementation details of the Modelica FMU Generator, the TinyOS FMU Generator, the Configuration Generator and the FMI Co-simulation Master are described as follows.

Modelica FMU Generator. If a SysML block is tagged with `ModelicaFMU`, HybridSim just needs to forward the location of the corresponding FMU to the Configuration Generator. If that block is tagged with `ModelicaSource`, HybridSim calls Dymola to compile the source codes and generate an FMU.

TinyOS FMU Generator. SysML blocks tagged with `TinyOSFMU` are processed similarly as blocks tagged with `ModelicaFMU`. For blocks tagged with `TinyOSSource`, HybridSim needs to compile nesC source codes, prepare the TinyOS Emulator, and compile the FMI Wrapper source codes.

To compile nesC source codes, HybridSim first checks how the TinyOS application is created. If it is directly imported from TinyOS libraries, HybridSim just needs to set up certain environment variables and read its associated *Makefile* to compile it. Otherwise, if it is created in the second way as described above, HybridSim will collect information about all the required TinyOS components and generate a *Makefile.* The output of this step is a set of ELF (Executable and Linkable Format) files, which will be loaded by the TinyOS Emulator.

The TinyOS Emulator is developed based on Avrora [42] in Java, which is extended as follows. Firstly, the TinyOS Emulator provides interfaces to input data into sensors from outside interactively. Secondly, the TinyOS Emulator can extract runtime values of variables from TinyOS components. By processing the ports tagged with `TinyOSVariable`, the TinyOS Emulator can infer the actual variable names in ELF files, and their memory addresses when these ELF files are loaded into RAM. Monitors are added to monitor these RAM addresses, which can reconstruct the variable values when they are updated. These two extensions enable the TinyOS Emulator to exchange data with a Modelica simulator. Finally, a new synchronization mechanism is developed, which can synchronize all threads of sensor motes, get information about the next system-wide event from local event queues, and enable the FMI Co-simulation Master to control the size of each synchronization step.

The FMI Wrapper is developed based on FMU-SDK [47] in C, which is extended with a more flexible mechanism to handle variable references. In addition, the FMI Wrapper provides convenient interfaces to interactive with the TinyOS Emulator through the Java Native Interface framework. Finally, the FMI Wrapper can process the output of the Configuration Generator to generate *modelDescription.xml* and compile all files into a stand-alone FMU with the TinyOS Emulator included.

Configuration Generator. The main output of this module is two configuration files for co-simulation. The first one is an Avrora configuration script, which specifies a set of options for the TinyOS Emulator, sensor types for each mote, and a set of variables whose values should be extracted and the TinyOS components in which they are declared. The second one is a connection configuration file, which specifies connections of the output and input ports of a TinyOS FMU and a Modelica FMU. This file is used by the FMI Co-simulation Master to exchange information between the TinyOS Emulator and a Modelica simulator.

FMI Co-simulation Master. Fig. 3.2 describes the algorithm of the FMI Co-simulation Master, which is developed based on PyFMI [48] in Python. The master first parses the connection configuration file and records connections in two hashtables. The key of an entry is an output port of the given FMU, and the value is its corresponding input port of the other FMU. After the two FMU slaves have been instantiated and initialized, the master begins to co-simulate them in steps. In each step, the master first reads the values of output ports of the Modelica slave, which are then forwarded to the TinyOS slave. Next, the step size δ is computed as the minimum of the specified communication step size, and the two incremental times of the next event in the TinyOS slave and in the Modelica slave, respectively. Therefore, no interaction event is missed. Then the master stimulates the TinyOS slave to emulate up to $t + \delta$. After that, the values of its output ports are forwarded to the input ports of the Modelica slave, which is then stimulated to simulate up to $t + \delta$ similarly. When the specified end time of co-simulation is reached, the master terminates both slaves and releases the resource occupied by them. The co-simulation results for both slaves are stored in Dymola format, which can be loaded into Dymola for further analysis, or plotted in HybridSim directly.

```
Input: Two FMUs, stopTime, connSize, config.txt
Output: TOSResult.txt, MdlResult.txt
 1: $[TOSConn, MdlConn] \leftarrow parse(config.txt)$;
 2: $TOSSlave \leftarrow loadFMU(TinyOSApp.fmu)$;
 3: $MdlSlave \leftarrow loadFMU(ModelicaSys.fmu)$;
 4: $TOSResult = newDymolaWriter(TOSSlave)$;
 5: $MdlResult = newDymolaWriter(MdlSlave)$;
 6: $TOSSlave.initialize()$; $MdlSlave.initialize()$;
 7: $t \leftarrow 0$;
 8: while $t < stopTime$ do
 9:     $values \leftarrow MdlSlave.get(MdlConn.keys)$;
10:     $TOSSlave.set(MdlConn.values, values)$;
11:     $\delta \leftarrow getStep(connSize, TOSSlave, MdlSlave)$;
12:     $TOSSlave.doStep(t, \delta)$;
13:     $values \leftarrow TOSSlave.get(TOSConn.keys)$;
14:     $MdlSlave.set(TOSConn.values, values)$;
15:     $MdlSlave.doStep(t, \delta)$;
16:     $TOSResult.writePoint()$; $MdlResult.writePoint()$;
17:     $t \leftarrow t + \delta$;
18: end while
19: $TOSSlave.terminate()$; $MdlSlave.terminate()$
20: $TOSSlave.free()$; $MdlSlave.free()$
```

Figure 3.2: FMI Co-simulation Master

3.5 Case Study

In this section, we use a comprehensive hydronic heating system as the case study to demonstrate the convenience and efficiency of HybridSim. Specifically, we investigate the impacts of packet loss and sampling rate that are introduced by wireless sensor networks on the heating system.

3.5.1 Modelica Model and TinyOS Application

Our Modelica model is developed based on an example provided by the Modelica Buildings Library [44], which comprehensively models the hydronic heating system for a building with energy storage and thermostatic radiator valves. Two rooms on the same intermediate floor are modeled using a dynamic model for the heat transfer through opaque constructions, with the same temperature above and below them. They share one common wall and have two windows. Realistic weather data traces from Chicago are fed into this model as the outside environment.

The hydronic heating system consists of a boiler, a storage tank and a radiator with a thermostatic valve in each room. The supply water temperature setpoint is reset based on the outside temperature. A three-way-valve mixes the water from the tank with the water from the radiator return. The pump has a variable frequency drive that controls the pump head. The building has a controlled fresh air supply with a heat recovery ventilator to preheat the outside air. Each room has a leakage model of the facade, through which the difference in air supply will flow if the supply and exhaust air are unbalanced.

A finite state machine is used to control the boiler and its pump. They are switched on when the temperature at the top of the tank is less than 1 Kelvin above the setpoint temperature for the supply water of the radiator loop. The boiler is switched on 10 seconds later than the pump. They are switched off when the temperature at the bottom of the tank reaches 55 °C. The pump is switched off 10 seconds later than the boiler.

In this case study, we only care about the temperatures of the two rooms, and the top and bottom of the tank, each of which is monitored by a temperature sensor. Originally, their readings are directly fed into the controller without any delay or loss. To investigate the impact of wireless sensor networks, the connections between the sensors and the controller are deleted. Instead, their readings are provided to four output ports respectively. Correspondingly, four input ports that are connected to the controller are created to get data from outside.

The TinyOS application consists of a set of relay motes and four sensor motes sampling the temperatures of the two rooms, and the top and bottom of the tank. In addition, a sink (i.e., base station) is assigned to collect all their readings through multi-hop communications. The Collection Tree Protocol [49] is selected as the routing protocol, and the Rician fading channel model is applied in the TinyOS Emulator.

Fig. 3.3 demonstrates a system configuration for one of our co-simulation scenarios. Four sensor motes (two on the left and two on the right) are created by referring to an existing ELF file, which is created by the nesC compiler. The sink and relay motes are created by referring to the *Makefile* for their source codes (the ELF file is actually compiled from this source codes), while the hydronic heating system model is imported from an existing Modelica FMU.

HybridSim first generates a configuration file for the TinyOS Emulator as

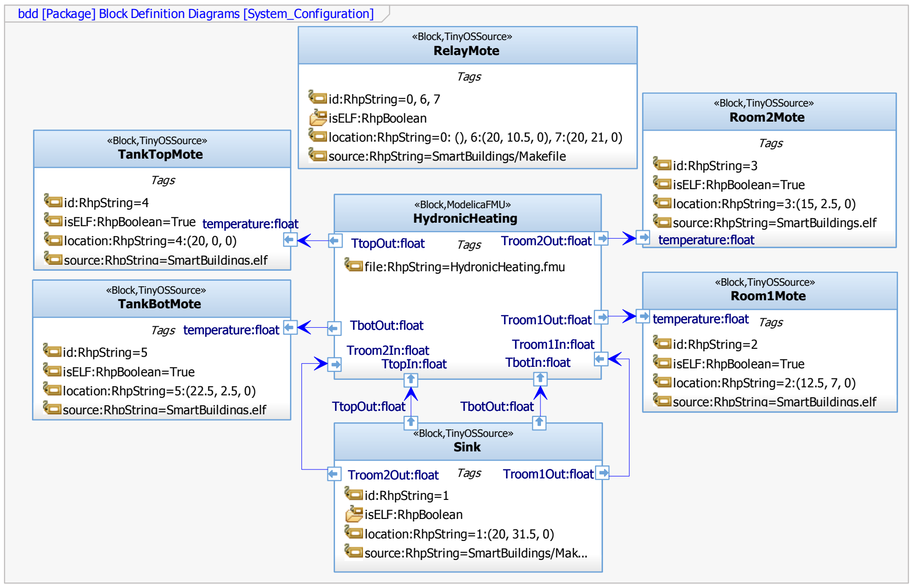


Figure 3.3: System Configuration

shown in Fig. 3.4. The topology file is generated by HybridSim based on the ID and location of each mote as specified in Fig. 3.3. Each parameter is assigned a unique reference number, which is required by the FMI standard. The name of an output parameter is `N⟨id⟩_⟨component⟩__⟨outPort⟩`, while that of an input parameter is `N⟨id⟩_⟨inPort⟩`. This file is also parsed to generate the TinyOS FMU. The corresponding configuration for co-simulation setup is shown in Fig. 3.5.

```
# options for the Avrora emulator
options = -nodecount=8 -topology-file=top.txt ... SmartBuildings.elf
# direction refNum paramName initialValue
output 915 N1_SmartBuildingsC__Troom1Out 20
output 925 N1_SmartBuildingsC__Troom2Out 20
output 935 N1_SmartBuildingsC__TtopOut 20
output 945 N1_SmartBuildingsC__TbotOut 20
input 715 N2_temperature 20
input 725 N3_temperature 20
input 735 N4_temperature 20
input 745 N5_temperature 20
```

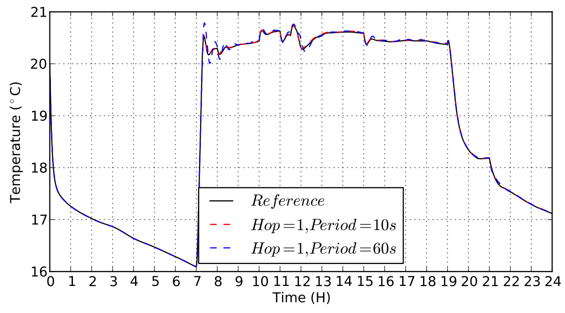
Figure 3.4: TinyOS Emulator Configuration

```
# Direction: output = input
# HH: HydronicHeating, SB: SmartBuildings
# Abbreviated from the original file
HH->TtopOut = SB->N4_temperature
HH->TbotOut = SB->N5_temperature
HH->Troom1Out = SB->N2_temperature
HH->Troom2Out = SB->N3_temperature
SB->N1_SmartBuildingsC__TtopOut = HH->TtopIn
SB->N1_SmartBuildingsC__TbotOut = HH->TbotIn
SB->N1_SmartBuildingsC__Troom1Out = HH->Troom1In
SB->N1_SmartBuildingsC__Troom2Out = HH->Troom2In
```

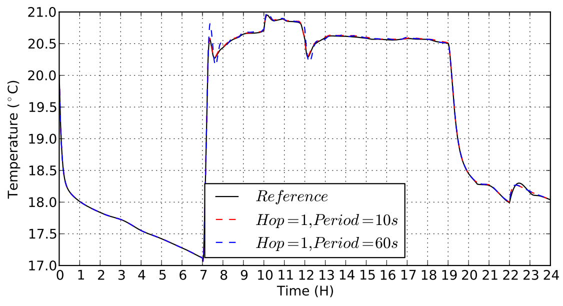
Figure 3.5: Co-simulation Configuration

3.5.2 Co-simulation Results

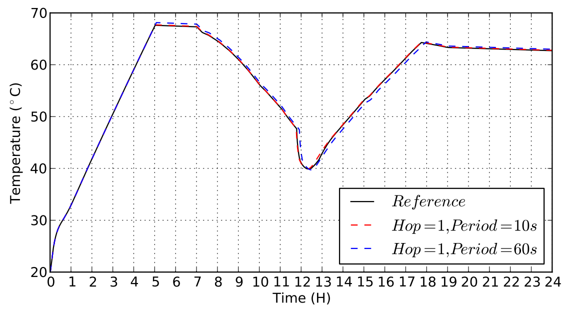
We first investigate the impact of sampling rate of sensor motes by simulating the system for one day. The communication step size for the TinyOS slave and the Modelica slave to exchange data and synchronize with each other is one second. Fig. 3.6 shows the simulation results when the sampling period is 10 seconds and 60 seconds, respectively. The four sensor motes locate within one-hop neighborhood of the sink, and no background traffic is considered. The simulation result of the original Modelica model in which all sensor readings are fed into the controller directly is used as the reference. As we can see, when the sampling period is 10 seconds, its impact is imperceptible. If the sampling period is increased to 60 seconds, there is a little variation, but its result is still very close to the reference. This is reasonable because temperature usually changes slowly. This indicates that wireless sensor networks can be used in some CPS to reduce deployment complexity and costs. Especially for slow response systems, such as our hydronic heating system, low sensor duty cycles can still guarantee acceptable system performance while increasing sensor lifetime significantly.



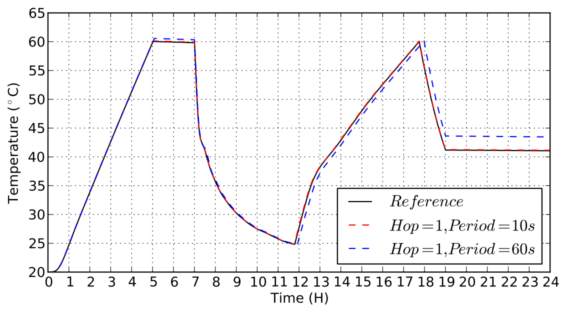
(a) Room1 Temperature



(b) Room2 Temperature



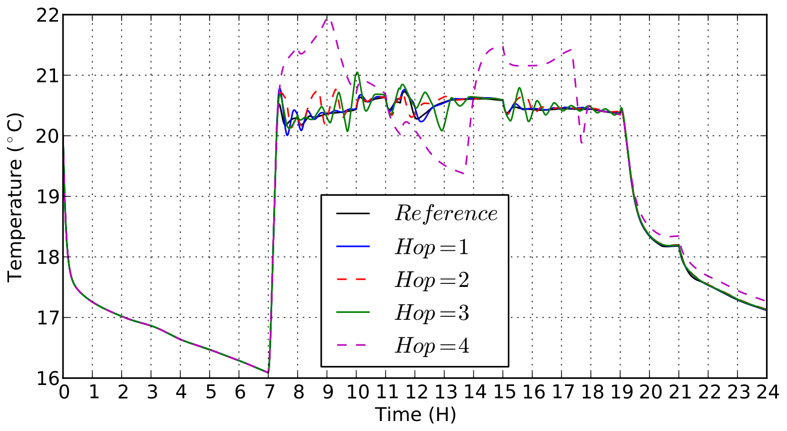
(c) Temperature of Tank Top



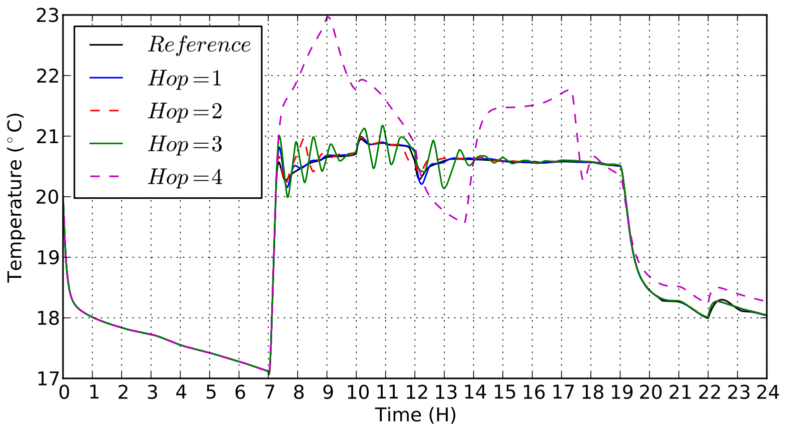
(d) Temperature of Tank Bottom

Figure 3.6: Impact of Sampling Rates

Smart Buildings, especially commercial buildings, are not clean environments for sensor communications, which usually suffer from heavy background traffic interference, such as WiFi traffic and Bluetooth traffic. In this scenario, we simulate the hydronic heating system under heavy WiFi traffic interference. Specifically, we feed a real-world noise trace into the TinyOS Emulator, which is measured in the Meyer Library at Stanford University with a large HTTP download and other WiFi traffic going-on [50]. The simulation results for different network sizes are shown in Fig. 3.7, which indicate that it is impractical to deploy a sensor network with more than 3 hops from a sensor mote to the sink in our case. Note that we only analyze the simulation results directly in HybridSim in this case study, which can also be processed by exiting tools for Avrora emulator and Modelica simulator.



(a) Room1 Temperature



(b) Room2 Temperature

Figure 3.7: Impact of Network Size

Table 3.1 shows the run-time efficiency of HybridSim. It can be seen that the performance of HybridSim depends on the network complexity. Actually, the efficiency is dominated by TinyOS Emulator that provides cycle-accurate instruction-level emulation.

Table 3.1: Run-time Efficiency (Simulation Time = 24 Hours)

Scenario	# of Nodes	Runtime (H)
1 Hop	6	4.54
2 Hop	8	6.18
3 Hop	11	9.04
4 Hop	14	11.24

3.6 Discussion

In this section, we discuss some observations, limits and future work for HybridSim. Communication step sizes impact co-simulation overhead and accuracy. HybridSim adopts variable communication step sizes based on the time of the next event in both slaves to provide fine-grained interactions between them, so that no event potentially requiring interactions is missed. However, in some scenarios, such a big overhead is not necessary. For example, in our case study, the TinyOS slave and the Modelica slave only interact at sampling points, and the sampling period is much larger than variable communication steps. If communication step sizes are fixed to a reasonable value (e.g., 1 second), the overhead imposed by the co-simulation master can be reduced significantly, while a good accuracy is still guaranteed. Systems engineers need to study the trade-off between overhead and accuracy depending on their system characteristics.

HybridSim can be extended with more interesting features. Firstly, the ns-3 network simulator [51] can be integrated into HybridSim. Since ns-3 is a more general discrete-event simulator for both wireless and wired networks, and preferable for fast prototyping of network designs, this feature will make HybridSim more interesting to a larger community. An FMI wrapper for ns-3 should be developed, similar to our FMI wrapper for the Avrora emulator. In addition, it will be very interesting to integrate Matlab/Simulink into HybridSim as well. For this feature, an FMU generator for Matlab/Simulink models is desired.

Another future work is to enhance the coupling between HybridSim and IBM Rational Rhapsody, which currently only serves as the environment for the FMI Co-simulation Master. We are interested in integrating the Rhapsody simulator into HybridSim, which can generate simulation source codes for SysML Activity Diagrams and Statechart Diagrams. With this feature, HybridSim will become a much better design framework and toolchain for model-based systems engineering. Another interesting topic is to support distributed co-simulations, which is also specified in the FMI standard and is more suitable for large complex systems.

3.7 Summary

In this chapter, we designed and implemented a modeling and co-simulation toolchain, called HybridSim, for Cyber-Physical Systems. HybridSim can transform and import existing TinyOS components and Modelica models into SysML, so that systems engineers can design and simulate overall systems in a uniform framework. Based on the FMI standard, HybridSim provides a robust and flexible mechanism to exchange data and synchronize the TinyOS Emulator and the Modelica simulator.

Therefore, HybridSim can leverage the advantage of their respective professional emulator and simulator, and investigate complex cyber-physical interactions. In addition, HybridSim enables domain engineering groups to work relatively independently, while facilitating systems engineers to design and evaluate overall systems by using outputs from domain groups. A comprehensive case study is discussed to illustrate the convenience and efficiency of HybridSim.

Chapter 4

Reduce System Analysis Complexity

4.1 Introduction

As systems engineers, we are intimately familiar with using graphical models to describe systems. However, these graphical models are non-unique and there is usually a wide range of behaviorally equivalent ways to model the same problem. One successful application of graphical models from a different community is Bayesian networks (see [52] for a review). In this chapter, we take some of the mathematical analysis of the graphical models of Bayesian networks and translate it into terms that are more familiar to systems engineers. In a systems oriented fashion, we may think of the Bayesian network as being a *subclass* of the more abstract class of commutative semirings, which has many other subclasses. As shown in Fig. 4.1, one particularly interesting subclass from the perspective of systems engineering is the Tropical semiring. It encodes optimization over structures where the overall cost function is the sum of costs over individual components.

The basic essence of each of these cases is to solve a problem described over a network of components where decisions in one component may affect the choices available in another component and there is a global objective that can only be understood by examining the complete space of decisions. Examples of this class of problem include vertex cover, independent set, dominating set, graph k-colorability,

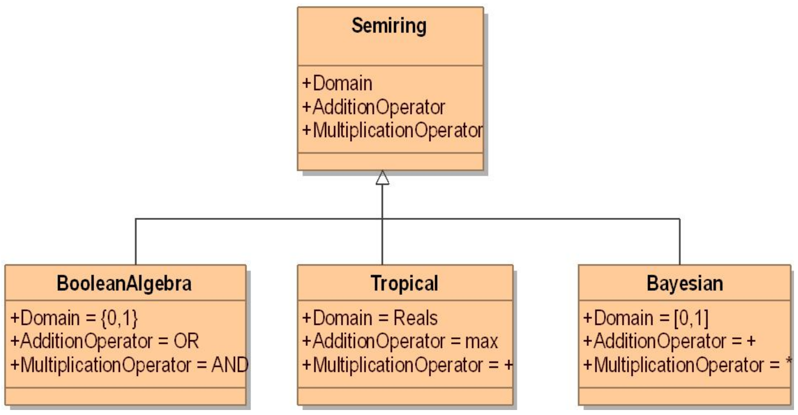


Figure 4.1: Interpretation of commutative semirings by subclasses

hamiltonian circuit, network reliability [53], and dynamic programming [54]. This class of problems is computationally challenging in general and embodies the curse of dimensionality. Using structural decomposition techniques of systems engineering is one approach towards solving these problems. However, there are very few tools available for doing this systematically. We present a tool that achieves this.

It turns out that complexity is exponential in *treewidth* and linear in problem size. The intuition behind this result is that problems on graphs are difficult to solve due to the presence of loops. Removing the loops by multiplexing variables (aggregating them into objects) can lead to tree decompositions of graph problems. Once the problem is in the form of a tree, then summary propagation is a viable technique for solving the problems. Multiplexing variables creates local complexity roughly in proportion to the number of variables tied together. More precisely, if we consider a discrete context, the space that needs to be explored is the product of the number of discretization bins, i.e., if there are N variables with D quantization levels each in an aggregate object, then the complexity of analyzing that object in D^N. The complexity of the overall system is the summation of the complexity of analyzing each system independently. This sum is dominated by the largest exponent in the system, which is precisely what the treewidth measures.

Our Contribution. In [55], we presented some theory of tree decomposition, which is summarized in Appendix A. This chapter describes a prototype that we have been working on to make the theory usable and several examples of problems solved using this tool. The main contribution of this chapter is an interactive tool for measuring treewidth of systems. A byproduct of this measurement is a system tree decomposition algorithm that can be used for analysis. We work out many examples using the tool and describe the algorithm used.

4.2 Related Work

Guenov [56] estimates the complexity to aid high-level designers in comparing alternatives during pre-competitive studies or during the architectural design process of composition systems. This approach is based on Boltzmann's entropy concept to measure the distribution of functional couplings in the system's decomposition. However, no mechanism is proposed to reduce the complexity.

Lu et al. [57] considered that the "overall difficulty" of an engineering system design consists of "inborn complication" due to custom requirements and external constraints as well as "acquired complexity" associated with uncertainty in satisfying the functional requirements caused by design decisions. They introduced the Axiomatic Design Theory and the Design-centric Complexity Theory to guide the creation and improvement of complex engineering systems. However, they cannot provide instant feedback on the impact of design decisions on the complexity.

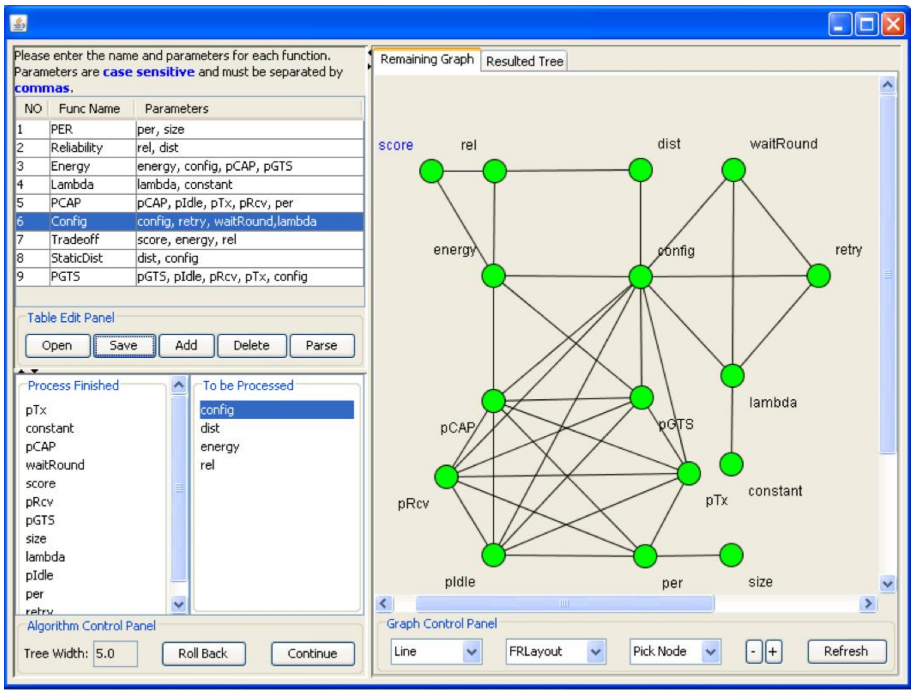
Clarke [58] presented a framework to reduce the complexity of temporal logic model checking in systems composed of many parallel processes by using additional interface processes to model the environment for a component. These interface processes are typically much simpler than the full environment of the component. By composing a component with its interface processes and then checking properties of this composition, one can guarantee that these properties will be preserved at the global level. However, this partitioning is ad-hoc and depends heavily on rules of thumb and the expertise of systems engineers.

A large number of model checking algorithms are based on the symbolic model checking method, which was first proposed in [59]. This method avoids building a state graph by using Boolean formulas to represent sets and relations. A variety of properties characterized by least and greatest fixed points can be verified purely by manipulations of these formulas using Ordered Binary Decision Diagrams. Instead of enumerating reachable states one at a time, the state space is traversed much more efficiently by considering large numbers of states at a single step. Such state space traversal is based on representations of state sets and transition relations as formulas, binary decision diagrams or other related data structures as in [60]. Several tools that can reduce the complexity of formal verification based on symbolic model-checking and homomorphic reduction are discussed in [61]. While these tools can reduce the complexity of formal verification efficiently, they cannot provide guidance on how to improve the system designs to facilitate formal verification further.

4.3 Tool Development and Case Studies

4.3.1 Tool Development

To facilitate the usage and enhance the understanding of the tree search algorithm, a user-friendly GUI was developed in Java, which enables users to control the execution of the algorithm interactively and view the results graphically. The GUI is shown in Fig. 4.2, painting the relationship graph for the parameters in our case study, which will be explained later.


Please enter the name and parameters for each function. Parameters are **case sensitive** and must be separated by **commas.**

NO	Func Name	Parameters
1	PER	per, size
2	Reliability	rel, dist
3	Energy	energy, config, pCAP, pGTS
4	Lambda	lambda, constant
5	PCAP	pCAP, pIdle, pTx, pRcv, per
6	Config	config, retry, waitRound,lambda
7	Tradeoff	score, energy, rel
8	StaticDist	dist, config
9	PGTS	pGTS, pIdle, pRcv, pTx, config

Process Finished: pTx, constant, pCAP, waitRound, score, pRcv, pGTS, size, lambda, pIdle, per, retry

To be Processed: config, dist, energy, rel

Tree Width: 5.0

Figure 4.2: GUI and the generated relationship graph for the case study

Function definitions can be loaded from a pre-saved file, or input to the table in the upper left corner, by specifying their names and parameters. Then they can be checked and parsed to the data structures used in the tree search algorithm. If all functions are defined correctly, the tree search algorithm will process the chordal vertices [55] automatically. The algorithm control area in the lower left corner will provide the list of unprocessed parameters and the parameters that have already been processed. Users can select an unprocessed parameter to continue the algorithm and the resulting treewidth will be calculated and updated incrementally. Users can also roll back the algorithm to its previous state and make a different choice, potentially with a smaller treewidth.

An observer thread is running in the background to update the relationship graph of the parameters and the resulted tree of cliques periodically, which are shown in the right tabbed panel. Users can also update them instantly by clicking the Refresh button. Based on the characteristics of the graph and the tree, users can select different layout algorithms to place the vertices automatically to get a better view, or arrange them manually. The Java Universal Network/Graph (JUNG) Framework[1] is used for data visualization.

4.3.2 Wireless Sensor Networks

We consider the trade-off analysis between energy efficiency and transmission reliability in wireless sensor networks, where the IEEE 802.15.4 standard is applied as the media access control protocol. For simplicity, we only provide high-level abstract functions here, emphasizing the abstract relationships between the parameters in each function. More details are available in [62]. The following functions are used in this trade-off analysis, in which the blue parameters are their outputs:

[1] `http://jung.sourceforge.net/index.html`

- **Tradeoff(score, energy, rel) = 0**. This function specifies the trade-off rules between energy efficiency and transmission reliability.

- **Reliability(rel, dist) = 0**. This function calculates the reliability based on the static distribution of the Markov chain model in [Wang et al., 2011], which models the peer-to-peer communications for time-critical applications in wireless sensor networks using the enhanced IEEE 802.15.4 protocol.

- **StaticDist(dist, config) = 0**. This function computes the static distribution, based on the configuration information specified for the protocol.

- **Config(config, retry, waitRound, lambda) = 0**. This function processes the protocol parameters, such as the maximum retransmission times and the maximum waiting rounds, to generate the configuration information. Lambda(lambda, constant) = 0. This function is defined to simplify the Config function, by processing other protocol-specific constants and feeding the results to the Config function.

- **Energy(energy, config, pGTS, pCAP) = 0**. This function calculates the expected energy consumption for each transmission, based on the configuration information, and the expected energy consumptions in the contention-based access period (CAP) and in the guranteed time-slot period (GTS).

- **PGTS(pGTS, config , pIdle, pRcv, pTx) = 0**. This function computes the expected energy consumption in the GTS period, based on the transmission power, receiving power, the power in the idle state and the configuration information.

- **PCAP(pCAP, pIdle, pRcv, pTx, per) = 0**. This function is very similar to the PGTS function, except that the packet error ratio (PER) is considered here.

- **PER(per, size) = 0**. This function simply calculates the PERs based on packet sizes.

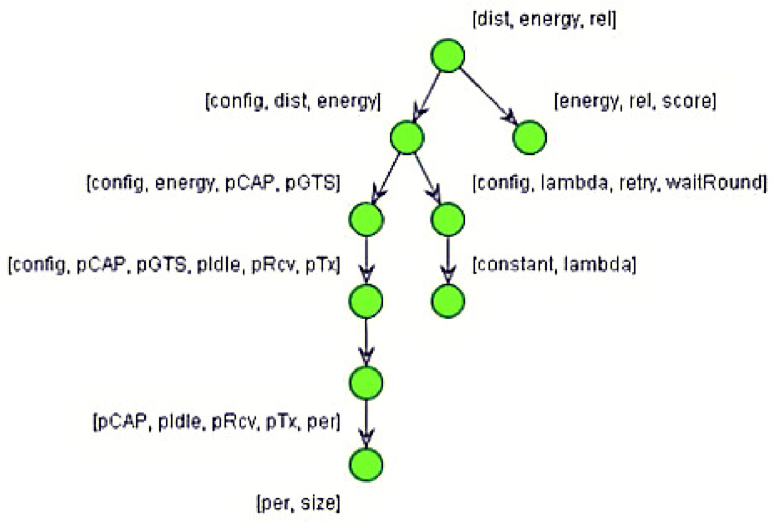

[dist, energy, rel]
[config, dist, energy]
[energy, rel, score]
[config, energy, pCAP, pGTS]
[config, lambda, retry, waitRound]
[config, pCAP, pGTS, pIdle, pRcv, pTx]
[constant, lambda]
[pCAP, pIdle, pRcv, pTx, per]
[per, size]

Figure 4.3: The generated tree of cliques

The generated tree of cliques is shown in Fig. 4.3, in which each vertex stands for a clique in the relationship graph of parameters, and the edge direction represents the reverse order of information propagation. When a vertex has received the information from all its children, it begins to calculate the parameters in its clique locally and propagate the result back to its parent. Now suppose every parameter can have 10 different values (continuous parameters can be sampled discretely). Then the complexity can be reduced significantly to: $10^2 * 2 + 10^3 * 3 + 10^4 * 2 + 10^5 + 10^6 = 1123200$, compared to 10^{16} in the original computation.

4.3.3 Quadrotor Example

Fig. 4.4 shows the relationships between variables in a quadrotor that is designed to fly out to a specified destination, land, perch and take observations. It uses a parametric diagram, which is exactly equivalent to a factor graph, in being a bipartite graph that has variable nodes in one partition and function nodes in the other partition. Consider the constraints shown in this diagram. The Tradeoff constraint reflects the fact that we are interested in the relationship between cost and range of the quadrotor. As indicated by the Cost constraint, the cost value is determined entirely, in this model, by the choice of payload and battery. The weight is also determined by these two variables, as shown by the Weight constraint. The range of the quadrotor, as indicated by the Range constraint, is determined by the choice of battery and the power requirements expressed as current. The flight current needed is determined by the weight of the quadrotor, as indicated by the Current constraint. Finally, there is a perch time variable that is solely determined by the payload as shown in the PerchTime constraint.

The fact that it is a factor graph means that summary propagation can be used as a solution algorithm with the correct interpretation of the summation and multiplication operations. This particular parametric diagram reflects a query on the tradeoff between range and cost. The constraint Tradeoff is a query in this case and modifies the structure of the parametric diagram, which in turn has an impact on the resulting tree decomposition. In working with this system of tree decompositions, this dependency of structure on the query occurs often. If a query relates two variables that were previously unrelated, then a link must be added to the graph reflecting this added coupling.

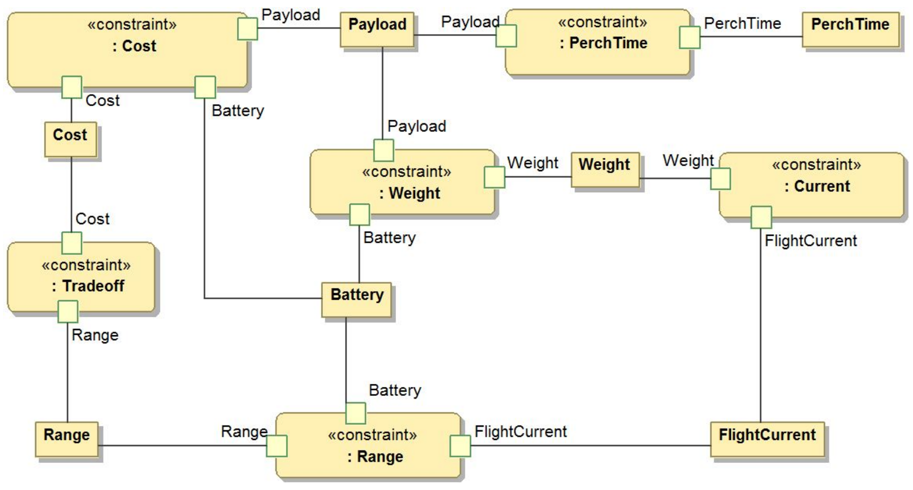


Figure 4.4: Parametric diagram for high level tradeoffs of a quadrotor

The different functions specify feasible regions of values for the various parameters, but there is a locality structure to this specification because certain variables are not directly related. For example, the current needed to fly the quadrotor depends on the weight (as indicated in the *Current* constraint, but these variables are not directly connected to the cost). Weight depends on the battery and payload chosen, which then directly contribute to the cost.

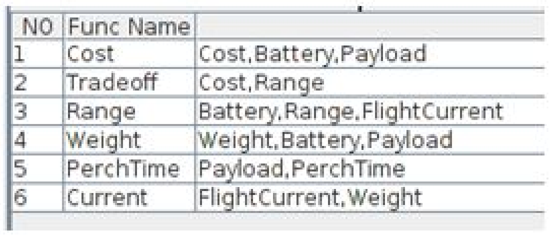
We would like to determine all feasible configurations with respect to range and cost in our trade study. We shall assume that every parameter takes on a discrete set of values, which could come from discretization. Naively, there are 7 variables in this system. Evaluating over all of them simultaneously using brute force could involve D^7 evaluations, where D is the number of discretization levels.

Fig. 4.5a shows the input to the tool representing the relationships between the variables. Compare this to Fig. 4.4. The name column contains exactly entries corresponding to the constraints of the parametric diagram and the parameter column contains the arguments to those constraints.

Fig. 4.5b shows the initial topology of the quadrotor (the functional dependence graph in the language of [55]), which is extracted from the relationships in Fig. 4.5a. Note that this graph is not chordal [55], which means that the designer will need to choose additional variable couplings for the system to decompose.

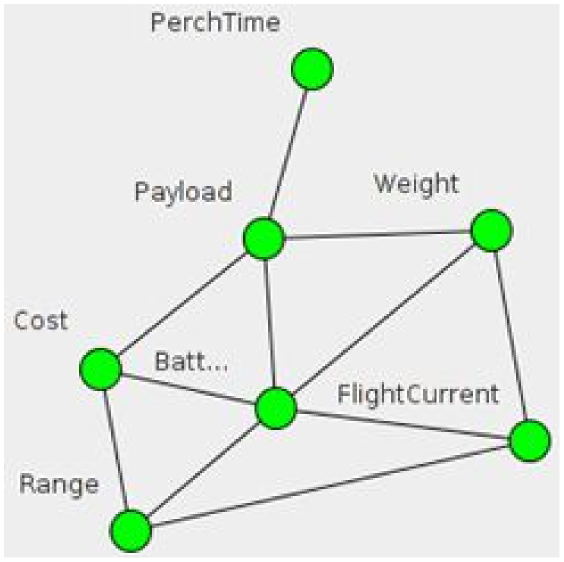
At this point, the designer has a decision to make because the only simplical node is PerchTime, which is eliminated by the algorithm. Elimination on the rest of the nodes creates fillins. To make this decision, the designer thinks about which variables most naturally fit together with respect to the fillins created. Since the relationship between weight and range is the most intuitive, the next elimination is FlightCurrent, which creates a fillin between weight and range. This is shown in Fig. 4.5c. One more fillin is neeed to complete the system decomposition.

A link was added between weight and range, coupling these two variables within the analysis even though there is no immediate equation describing this relationship. This is an artifact of performing the tree decomposition of the system. Of the remaining variables, the next most intuitive relationship is the one between payload and range, so we eliminate cost next, which requires payload and range to be coupled. Fig. 4.5d shows the result. This system is chordal and has a tree decomposition. The tree structure consists of three tetrahedrons that are stacked next to each other and a tail consisting of the PerchTime, which is only loosely coupled with the rest of the system. The tool produces Fig. 4.5e as the tree decomposition of the system using these hints from the designer.

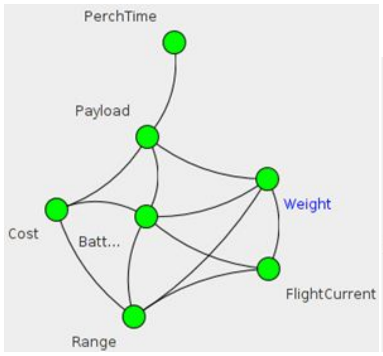


NO	Func Name	
1	Cost	Cost,Battery,Payload
2	Tradeoff	Cost,Range
3	Range	Battery,Range,FlightCurrent
4	Weight	Weight,Battery,Payload
5	PerchTime	Payload,PerchTime
6	Current	FlightCurrent,Weight

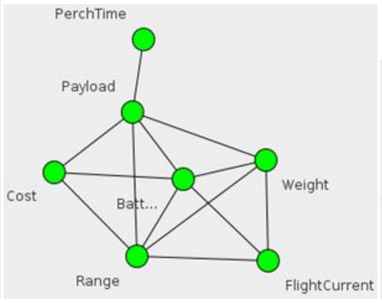
(a) Input to the tool



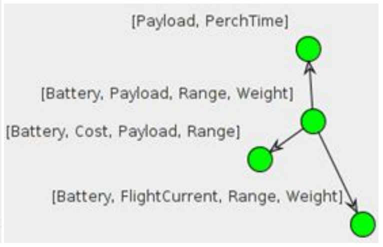
(b) Functional dependence graph (initial)



(c) Functional dependence graph (with additional fillins)



(d) Chordal transformation


[Payload, PerchTime]
[Battery, Payload, Range, Weight]
[Battery, Cost, Payload, Range]
[Battery, FlightCurrent, Range, Weight]

(e) Join tree

Figure 4.5: Quadrotor example

The last step in the analysis described in [55] is to map the original constraints and functions back to the resulting join tree. The most natural language for expressing this is a block diagram, as shown in Fig. 4.6. The associations between blocks are labeled according to the shared variables. There is always a way to assign the constraints back to the aggregations in such a way that every constraint has all its parameters in its local block. Though mapping is not unique in general, it happens that the assignment of the constraints back to the structure is unique in this case.

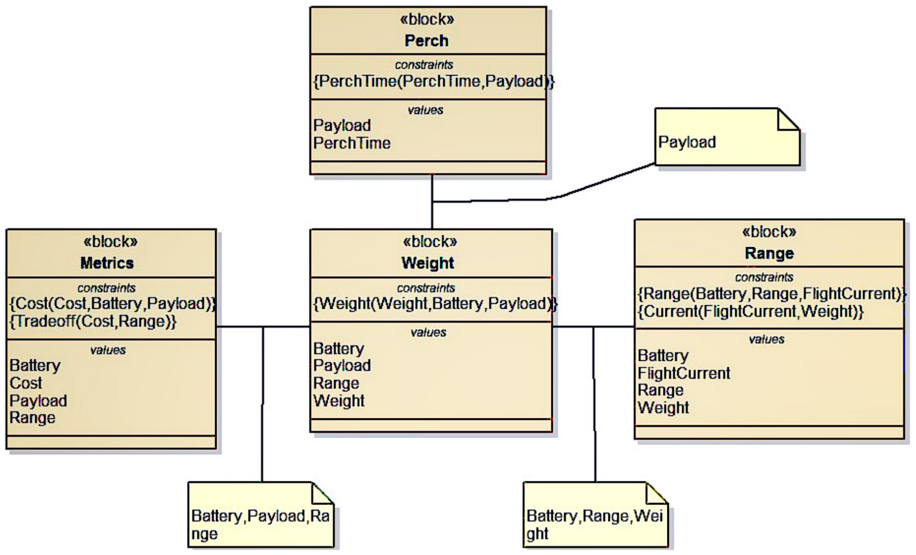


Figure 4.6: The completed Block Diagram of the tree decomposition of the quadrotor

To analyze the system shown in Fig. 4.6, we use a very simplistic algorithm using sets. Each block, Perch, Metrics, Weight, and Range, can be thought of as describing a set of feasible points based on the constraints. The overall space of the system can be described as the intersection of the spaces described in the blocks. To apply summary propagation, we use set intersection as the multiplication operation and projection as the summation operation. Since this is a trade study, our goal is to evaluate the Metrics block. Fig. 4.7 depicts the general strategy of evaluation. Using the decomposition of Fig. 4.6 reduces the complexity of analyzing the system from D^7 down to $3D^4 + D^2$. This is a significant reduction. Suppose, for example, we use a grid of 20 points. $20^7 = 1.28 * 10^9$ while $3 * 20^4 + 20^2 = 480,400$, which is orders of magnitude fewer samples.

Now we summarize how systems engineers can use our tool to improve their designs. Firstly, a system engineer transforms the constraints (i.e., functions) in the SysML Parametric Diagrams and inputs them into our tool as in Fig. 4.5a. If the generated functional dependence graph is not chordal (e.g., Fig. 4.5b), the engineer needs to interact with our tool to help it transform the initial functional dependence graph into a chordal graph, which is essential for our algorithm. The basic operation is to add more fillins. However, the transformation is not unique. Different set of added fillins can result in different expected complexity of system analysis, which is dominated by the size of the maximum clique in the chordal graph. The engineer can try different options, and if the current selection induces a large complexity, the engineer can roll back the current graph to its previous status and continue investigation with other options. Therefore, our tool can expose a sequence of design choices to systems engineers to provide instant feedback about the influence of a design decision on the complexity of system analysis.

With a chordal graph, our tool will generate a join tree, based on which the engineer can create a SysML Block Diagram by assigning constraints to blocks. This Block Diagram is essentially a factor join tree, in which blocks are the factor nodes, and the intersection of parameters in two blocks is the variable node between them, as shown in Fig. 4.6. Based on this Block Diagram, the engineer can revise the original SysML Parametric Diagrams accordingly, such that constrains and parameters

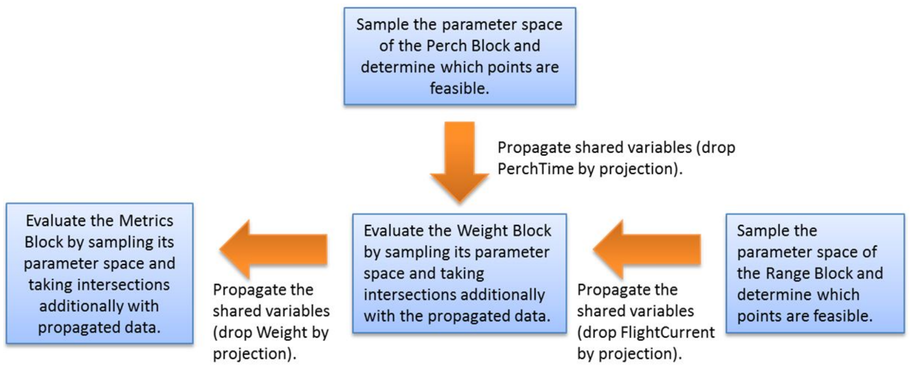

Sample the parameter space of the Perch Block and determine which points are feasible.
Propagate shared variables (drop PerchTime by projection).
Evaluate the Metrics Block by sampling its parameter space and taking intersections additionally with propagated data.
Propagate the shared variables (drop Weight by projection).
Evaluate the Weight Block by sampling its parameter space and taking intersections additionally with the propagated data.
Propagate the shared variables (drop FlightCurrent by projection).
Sample the parameter space of the Range Block and determine which points are feasible.

Figure 4.7: Summary propagation applied to the block diagram of Fig. 4.6. We treat each of the blocks as sets. The overall system is understood as the intersection of all the sets. We can use a generalized version of summary propagation to efficiently run queries on this structure

in the same block can be grouped together locally. Then the engineer can analyze the system using summary propagation as shown in Fig. 4.7. Therefore, our tool can provide guidance for systems engineers to improve their designs and analyze their systems.

Other examples are available in [63].

4.4 Discussion

One interesting property of the technique is how counterintuitive these join trees are from the perspective of creating block diagrams. However, looking at Fig. 4.5d reveals an interesting relationship between the geometry of the chordal decomposition and the resulting block structure. The Battery and Range variables are shared by the three blocks. It is apparent in the geometry that these two variables form an axis which connects the three tetrahedrons and thus Battery and Range are shared variables over three of the blocks in the block diagram. We are not accustomed, as engineers to expressing decompositions using shared variables, although it is apparent that this is natural because the constraint structure has a both locality and a dependence structure. Having a tool for performing this analysis certainly helps in finding the tree decompositions.

As shown through the examples, this is a very general technique that can be applied to many domains. In the examples of this chapter, the sets are static in nature. In [64], we show how the same technique of composition and projection can help in the formal analysis of dynamic Bayesian networks.

4.5 Summary

We have presented a tool that uses an interactive method to compute junction trees and show how the technique can be applied in structuring the analysis of broad range of systems. Theoretically, problems that can be encoded as commutative semirings are amenable to analysis by this technique, but it is not limited to this domain. We believe this tool and graphical decomposition technique could be of use to many systems engineers. It is complexity aware and generates decompositions that are amenable to localized computational analysis.

Part II

Flash-based Data Storage and Incremental ϵ-Approximate Querying

Chapter 5

Distributed Database System for Wireless Sensor Networks

In the second part of the dissertation, we describe the design and implementation of an energy-efficient distributed database system that enables sensors to store readings *in situ* in local flash memories and supports incremental ϵ-approximate querying. This system can greatly simplify the development of sensor network applications by abstracting data collection and processing. This system is integrated into the Service Model Library in WSNDesign.

In this chapter, we introduce our main contributions in the second part of the dissertation, formulate the problem, and describe our design considerations. In Chapter 6, we present the detailed design of HybridStore, including the index structure and query processing algorithm. HybridDB is described in Chapter 7, including the incremental query processing algorithm and adaptive error distribution mechanism.

5.1 Introduction

One of the main challenges in wireless sensor networks is the storage and retrieval of sensor data. Traditional centralized data acquisition techniques (e.g., [65]) suffer from large energy consumption, as all the readings are transmitted to the sink. In long-term deployments, it is preferable to store a large number of readings *in situ* and transmit a small subset only when requested [8, 9]. This framework becomes practically possible with the new generation NAND flash memories that has dramatically altered the capacity and energy-efficiency of local storage. Recent studies show that NAND flash memories is at least *two orders of magnitude cheaper* than communication and *comparable in cost* to computation [10]. Therefore, extending NAND flash memories to off-the-shelf low-end sensor platforms can potentially improve in-network processing and energy-efficiency substantially.

However, due to the fundamentally different read and write semantics of NAND flash memories, and tightly constrained resource on sensor platforms, designing an efficient resource-aware data management system for flash-based sensor devices is a very challenging task. Existing techniques, such as LA-Tree [11], μ-Tree [12], B-File [13], FlashDB [14] and PBFilter [15], are not applicable to sensor platforms due to their large RAM footprints. Capsule [66] provides a stream-index object to store data stream efficiently, however, with very limited supports for general queries. Other works, such as TL-Tree [16] and FlashLogger [17], can only process simple time-based queries. More importantly, however, none of existing works take advantage of both the on-board *random-accessible* NOR flash memories that is quite suitable for index structures available in current sensor platforms, and external economical energy-efficient NAND flash memories with high-capacity, which is ideal for massive data storage.

In the network level, data approximation is a popular technique to support in-network energy-efficient query processing to retrieve multi-dimensional readings from multiple sensor motes. Traditional data approximation methods [18–20, 67] require users to specify fixed error bounds to address the trade-off between result accuracy and energy efficiency of queries. However, in many real-world scenarios, it is unfeasible and inefficient for users to determine in advance what error bounds can lead to affordable cost or acceptable results. On one hand, if an error bound is too tight, much energy will be wasted to retrieve more readings than needed, resulting in an over-qualified result. On the other hand, if the error bound is not tight enough, the set of readings returned by the query cannot produce a satisfactory result. In this case, the user needs to re-issue this query with a tighter error bound. Traditional schemes will treat it as an independent new query, and thus all readings that have been retrieved by the previous query will be transmitted again, resulting in much energy waste as well. To handle these problems, we need an incremental approximate querying mechanism.

In the second of part of the dissertation, we design and implement HybridDB, an efficient light-weight distributed database system for flash-based storage-centric wireless sensor networks. HybridDB exploits a novel efficient resource-aware data storage system, called HybridStore, which exploits both the on-board NOR flash and external NAND flash to store and query readings *in situ* on each sensor mote. In order to completely avoid expensive in-place updates and out-of-place writes to existing NAND pages, the index structure is created and updated in the NOR flash. To handle the problem that the capacity of NOR flash on low-end sensor platforms is very limited (512KB to 1MB), HybridStore divides a sensor data stream into segments, the index of which can be stored in one or multiple erase blocks in the NOR flash. Since the NAND flash is much faster and more energy-efficient for reading, the index of each segment is copied to the NAND flash after the NOR segment is full. Therefore, all NAND pages used by HybridStore are fully occupied and written in a purely sequential fashion, which means it can support both raw NAND flash chips and FTL-equipped flash packages efficiently.

HybridStore can process typical joint queries involving both time windows and key value ranges as selection predicates extremely efficiently even on large-scale datasets, which sharply distinguishes HybridStore from existing works. The key technique is a novel index structure that consists of the inter-segment skip list, and the in-segment β-Tree and Bloom filter of each segment. The inter-segment skip list can locate the desired segments within the time window of a query efficiently. The β-Tree of a segment exploits a simple prediction-based method to split each node in the tree adaptively to generate a rather balanced tree, even when key values are very unevenly distributed. The Bloom filter of a segment facilitates value-based *equality* queries inside that segment, which can detect the existence of a given key value efficiently. Our index can eliminate a substantial number of unnecessary page reads when processing joint queries.

In addition, HybridStore can trivially support time-based data aging without any extra overhead, because no garbage collection mechanism is needed here, which can induce extensive page reads and writes to move valid pages within the reclaimed erase blocks to new locations. Finally, HybridStore provides a simple and efficient failure recovery mechanism that can guarantee the highest level of data consistency without the need for any checkpoint.

Based on HybridStore, HybridDB provides the support for incremental ϵ-

approximate querying that enables clients to retrieve a just-sufficient accuracy level of sensor data by issuing sub-queries with decreasing error bounds. Unlike traditional approximate querying methods, sensor readings that have been already retrieved by previous sub-queries with looser error bounds will not be transmitted again. HybridDB processes a sub-query with maximum absolute (L_∞-norm) error bound ϵ_i ($\epsilon_i \geq 0$) with the help of a proxy in two steps. Firstly, the proxy forwards the sub-query to the sensors but with error bound $\epsilon_{i,1} \leq \epsilon_i$. Each sensor processes this sub-query, retrieves the incremental set of readings from HybridStore by applying temporal approximate locally with error bound $\epsilon_{i,1}$, and transmits them to the proxy. Secondly, after the proxy has received readings from all sensors, it applies spatial approximate on them with error bound $\epsilon_{i,2}$, and returns the incremental set of readings to the client. We prove that the L_∞-norm error of the set of readings received by the client is guaranteed to be bounded by $\epsilon_{i,1} + \epsilon_{i,2}$. Thus, if HybridDB chooses $\epsilon_{i,2} = \epsilon_i - \epsilon_{i,1}$, the overall error bound of the sub-query from the client is satisfied. Traditional proxies are pre-deployed more powerful gateways. However, as smart phones have become ubiquitous and powerful mobile computing platforms [68, 69], we exploit them as temporary proxies.

For the following sub-query, if its error bound ϵ_{i+1} ($\epsilon_{i+1} < \epsilon_i$) satisfies $\epsilon_{i+1} \geq \epsilon_{i,1}$, the proxy can process it directly using the readings that have been retrieved in previous sub-queries, without retrieving any new readings from sensors. The incremental set of readings that can improve the error bound of spatial approximate from $\epsilon_{i,2}$ to $\epsilon_{i+1} - \epsilon_{i,1}$ will be returned to the client directly. Therefore, for each sub-query, the value of $\epsilon_{i,1}$ decides the trade-offs of energy consumption between sensor motes and the proxy, and of response times between the current sub-query and the following sub-queries. Smaller $\epsilon_{i,1}$ means more readings must be retrieved from sensors, and thus larger response time for the current sub-query. However, the proxy may be able to satisfy the error bounds of following sub-queries using only the readings buffered locally with higher probability, and thus reduce their response times significantly. To balance these trade-offs, HybridDB adopts an adaptive error distribution mechanism between $\epsilon_{i,1}$ and $\epsilon_{i,2}$, based on the parameters of each sub-query and the distribution of sensor readings.

Our implementation of HybridDB in TinyOS 2.1 consumes approximately 22.5KB ROM and 3.76KB RAM, which is well below the limit of most constrained sensor platforms. We conduct both simulations with a large-scale real-world dataset and testbed experiments to investigate the performance of HybridDB. Our evaluation results demonstrate that HybridDB can process ϵ-incremental approximate queries with both time windows and value ranges as selection predicates efficiently and balance the trade-offs effectively.

5.2 Related Work

In this section, we first review existing works on flash-based storage systems, and then discuss prior approximate data retrieval techniques.

5.2.1 Flash-based Storage Systems

A large number of flash-based storage systems have been proposed in the last few years, for both resource-constrained embedded systems and high performance platforms. The energy efficiency of currently available flash-based storage options for sensor platforms is comprehensively evaluated in [10], which showed that NAND flash is at least two orders of magnitude cheaper than communication and comparable in cost to computation. Their results introduced a new dimension in traditional computation-communication trade-offs and a promising option for the design of energy-efficient sensor networks and data-centric applications. Later, they proposed Capsule [66], a log-structured object storage abstraction for flash-based sensor platforms. Capsule provides a programming-oriented framework consisting of commonly used storage objects such as streams, files, arrays, queues and lists to facilitate the development of storage-centric sensor network applications. Another generic flash-based file system is presented in [70], which consumes vary small RAM resource and provides a POSIX-compatible programming interface to hide the complexity of various flash memory operations. However, none of these works can support efficient data retrieval and query processing on historical information, for which various challenges and querying capability requirements are discussed in [8].

FlashDB [14] is a self-tuning B^+-tree based index that dynamically adapts its storage structure to the mix of reads and writes workload. However, its RAM footprint increases fast as the number of readings increases, which makes it only suitable for gateways. FlashLogger [17] incorporates a suite of compression algorithms suitable for progressively compressing time series scalar, audio, and image data to provide data logging service with amnesic compression in a flash-efficient manner. However, it can only support time-range queries. The interplay between RAM and flash memory is explored in [71], which proposed a memory-adaptive storage system. Although they have compared a partition-based index with a global index, they did not investigate how to take advantage of the partition-based index and organize partitions efficiently. As a result, the RAM footprint of their approach is large because the per-partition B-tree index, interval table, and the *last page of each interval's list* must be maintained in RAM. TL-Tree [16] is designed to minimize out-of-place writes to NAND flash by making use of program flash memory on sensor platforms, which complicates the storage management at the risk of messing up the installed program. Besides, TL-Tree only supports time-range queries as well.

The most related works are Antelope [1] and MicroHash [9]. Antelope is a light-weight database management system for low-end sensor platforms based on the Coffee file system [70], which enables run-time creation and deletion of databases and indexes. However, its main index for value-based queries, MaxHeap, requires expensive byte-addressable random writes in flash. Therefore, Antelope is more suitable for NOR flash, which limits its performance because NOR flash is much slower and more energy-consuming compared to NAND flash. In addition, it can only retrieve *discrete values* in value-based range queries. MicroHash is an efficient index structure for NAND flash-based sensor devices, supporting value-based equality queries and time-based range queries *separately.* However, it suffers from out-of-place writes to existing pages, resulting in long chains of partially occupied pages. They alleviated this problem by combining multiple such pages into a fully occupied page, which induces extensive page reads and writes during insertions. More importantly, neither Antelope nor MicroHash can support joint queries involving both time windows and value ranges as selection predicates efficiently. That means, even though a query just wants to search readings within a certain value range in a small time window, they still needs to traverse the *whole global* index.

Other works designed flash-based storage systems for high performance platforms to overcome the access bottlenecks of hard disks. B-File [13] is designed to efficiently maintain a large sample in a flash memory and query them for a subsample of an arbitrary size. They also investigated the performance of sequential, random and semi-random writes on FTL-equipped flash packages. LA-Tree [11] minimizes accesses to flash by performing update operations in a lazy manner using cascaded buffers to amortize the cost of node reads and writes. The buffer sizes are dynamically adapted to workload using an online algorithm. PBFilter [15] organizes the index structure in a purely sequential way using two principles called summarization and partitioning based on Bloom filters to lookup keys efficiently. μ-Tree [12] minimizes the number of cascaded flash page writes when a leaf node is updated by putting together all the nodes along the path from the root to the leaf into a single flash page. SkimpyStash [72] uses a hash table directory in RAM to index key-value pairs stored in a log-structured manner on flash, where multiple keys that resolve to the same hash table bucket are chained in a linked list. SkimpyStash can only support key equality lookups as PBFilter. None of these works can be applied to low-end sensor platforms due to their large RAM footprints.

5.2.2 Approximate Data Retrieval

Approximate data retrieval is an efficient way to reduce energy consumption in wireless sensor networks. [73] proposed an optimal online algorithm for constructing a piecewise constant approximation of a time series produced by a single sensor, guaranteeing a L_∞-norm error bound. [74] presented distributed regression to build in-network models of sensor data based on kernel linear regression. Instead of transmitting raw readings, sensors communicate constraints on the model parameters. Then each sensor can answer queries for its local region based on the data model. BBQ [19] exploits a similar approach to build statistical models to reduce the sensing and communication cost for processing queries with user-specified error intervals and confidence levels. Later, they designed MauveDB [75], a database system integrating statistical models to support model-based views that provide independence from the details of the underlying data generating mechanism and hide the irregularities of the data by using models to present a consistent view to the users.

Ken [18] uses replicated dynamic probabilistic models at both the base station and sensor motes to reduce communication cost for continuously obtaining approximate data from the sensor network, by intelligently exploiting spatial correlations across sensor nodes. Region Sampling [76] tries to support approximate data retrieval in a different way, by segmenting a sensor network into partitions of non-overlapping regions and performing sampling and local aggregation for each region to bound the energy consumption. Although Region Sampling can be used to incrementally retrieve data from sensor networks, the full dataset cannot be recovered from these samples with bounded L_∞-norm error. TSAR [20] provides a hierarchical architecture to separate data from metadata by employing local archiving at the sensors and distributed indexing at the proxies. TSAR employs a multi-resolution ordered distributed index structure at the proxy tier for efficiently supporting spatio-temporal and value queries, and at the sensor tier supports energy-aware adaptive summarization that can balance the trade-off between the cost of transmitting metadata to the proxies and the overhead of false hits due to a coarse-grain index .

EAQ [77] is the work most related to our incremental ϵ-approximate querying mechanism. EAQ exploits a data shuffling algorithm to convert a dataset into a multi-version array that enables users to incrementally refine previously obtained approximate data to reach arbitrary accuracy and recover approximate versions of the entire dataset. Although a modified version of their data shuffling algorithm is exploited in our system, HybridDB exhibits three major differences from EAQ. Firstly, EAQ can only handle a snapshot dataset that consists of the readings generated by all sensors in the same period, while HybridDB can process historical querying with large time windows efficiently. Secondly, EAQ can only support simple refinement queries on a snapshot dataset, while HybridDB can process both complicated refinement queries and zoom-in queries to search interesting events efficiently. Finally, HybridDB exploits an adaptive error distribution mechanism to balance trade-offs, which is not considered in EAQ.

5.3 Problem Formulation

In this section, we formulate the approximate error model, incremental ϵ-approximate querying and sensor data storage, and describe our research objectives.

5.3.1 Query Model and Data Storage

We consider a wireless sensor network consisting of N motes deployed in a small geographical area. The network performs data sampling once every τ seconds. For each sampling period, every mote measures the environment and generates a reading $r_{id,t} = \langle id, t, key, v_2, \ldots, v_d \rangle$, where id is the identifier of this mote, t is the timestamp when this reading is generated (can be either the elapsed time since mote id is booted, or the epoch number), key is the attribute that will be indexed, and $v_2, \ldots, v_d$, are other attributes observed by this mote. The ID differences reflect the geographical proximity of sensor motes. In other words, nearby sensor motes are assigned closer IDs, and vice versa. This can be achieved by assigning manually as in our testbed, or mapping automatically as in [77], which is beyond the topic of this paper. Note that a large sensor network can be considered to consist of multiple such small geographical areas.

Given a time interval $[t_1, t_2]$, the set $\{r_{id,t}\}_{t=t_1}^{t_2}$ is called the *local dataset* generated on mote id, and $\Upsilon = \{r_{id,t}\}_{id=1,\ t=t_1}^{N,\ t_2}$ is called the *network dataset.* An approximate version $\widetilde{\Upsilon}$ of Υ conceptually contains a corresponding reading $\tilde{r}_{id,t}$ for each $r_{id,t}$ (i.e., $\widetilde{\Upsilon} = \{\tilde{r}_{id,t}\}_{id=1,\ t=t_1}^{N,\ t_2}$), but has smaller actual representation. The maximum absolute (L_∞-norm) error of the approximation $\widetilde{\Upsilon}$ is defined as:

$$L_\infty(\widetilde{\Upsilon}, \Upsilon) = \max_{id=1}^{N} \max_{t=t_1}^{t_2} \parallel (\tilde{r}_{id,t} - r_{id,t}) \times w \parallel_\infty = \max_{id=1}^{N} \max_{t=t_1}^{t_2} \max_{j=1}^{d+2} |(\tilde{r}_{id,t}[j] - r_{id,t}[j]) \times w_j|$$

where $\tilde{r}_{id,t}[j]$ and $r_{id,t}[j]$ are the j^{th} field of $\tilde{r}_{id,t}$ and $r_{id,t}$, respectively, and w_j ($0 \leq w_j \leq 1$) is the weight of this field.

An approximate query $Q = \{\rho_1, \rho_2, \ldots, \rho_\lambda\}$ consists of λ sub-queries. Each sub-query $\rho_i = \{[t_{i,1}, t_{i,2}], [k_1, k_2], \epsilon_i\}$ retrieves a subset O_i of $\Psi_i = \{r_{id,t} \mid (t_{i,1} \leq t \leq t_{i,2}) \wedge (k_1 \leq key \leq k_2)\} \subseteq \Upsilon$ such that the approximate version $\widetilde{\Psi}_i$ of Ψ_i recovered from O_i can guarantee $L_\infty(\widetilde{\Psi}_i, \Psi_i) \leq \epsilon_i$. To simplify the statements, we also denote $L_\infty(O_i) \triangleq L_\infty(\widetilde{\Psi}_i, \Psi_i)$. Here, we require that $0 \leq \epsilon_{i+1} < \epsilon_i$ and $[t_{i+1,1}, t_{i+1,2}] \subseteq [t_{i,1}, t_{i,2}]$ ($i = 1, 2, \ldots, \lambda - 1$). If $[t_{i+1,1}, t_{i+1,2}] = [t_{i,1}, t_{i,2}]$, ρ_{i+1} is called a *refinement* to ρ_i; if $[t_{i+1,1}, t_{i+1,2}] \subset [t_{i,1}, t_{i,2}]$, ρ_{i+1} is called a *zoom-in* to ρ_i. Refinement and zoom-in sub-queries enable clients to retrieve sensor data gradually. A client can first obtain an overall view of the data by issuing ρ_1 with large ϵ_1 to learn the general situation, and then issue refinement sub-queries to get more details gradually. If interesting phenomena are observed in some time interval, the client can issue zoom-in sub-queries to focus on more details for that time interval. Therefore, the client can find interesting events efficiently without retrieving unnecessary readings from sensors, resulting in significant energy saving and much shorter response time.

The key technique to support refinement and zoom-in sub-queries is *incremental data retrieval.* For each ρ_i, instead of treating it as an independent sub-query and returning O_i as in traditional approximate query processing mechanisms, HybridDB processes it as an intermediate step of Q and returns only $\Delta_i = O_i \setminus O_{i-1}$, which is called the *incremental set of readings* for ρ_i. The client can construct O_i from $\Delta_1, \Delta_2, \ldots, \Delta_i$, because $O_i \subseteq \bigcup_{j=1}^{i} \Delta_j$. If every ρ_j $(j = 1, 2, \ldots, i)$ is a refinement sub-query, $O_i = \bigcup_{j=1}^{i} \Delta_j$. Otherwise, $\bigcup_{j=1}^{i} \Delta_j$ on the client side contains readings outside $[t_{i,1}, t_{i,2}]$. In this case, O_i can be constructed by filtering all $r_{id,t} \in \bigcup_{j=1}^{i} \Delta_j$ satisfying $t \notin [t_{i,1}, t_{i,2}]$.

Based on the above notations, *incremental ϵ-approximate querying* is formulated as follows:

Definition 5.1 (Incremental ϵ-approximate Querying)**.** HybridDB is said to support *incremental ϵ-approximate querying* if for any query $Q = \{\rho_1, \rho_2, \ldots, \rho_\lambda\}$, where $\rho_i = \{[t_{i,1}, t_{i,2}], [k_1, k_2], \epsilon_i\}$ $(i = 1, 2, \ldots, \lambda)$ with $0 \leq \epsilon_{i+1} < \epsilon_i$ and $[t_{i+1,1}, t_{i+1,2}] \subseteq [t_{i,1}, t_{i,2}]$ $(i = 1, 2, \ldots, \lambda - 1)$, all the following conditions are satisfied:

1. For any ρ_i $(i = 2, \ldots, \lambda)$, only $\Delta_i = O_i \setminus O_{i-1}$ are returned to the client.

2. $\Delta_1 = O_1$ and $\Delta_i \cap \Delta_j = \emptyset$ for all $i \neq j$ $(i, j = 1, 2, \ldots, \lambda)$.

3. For any ρ_i $(i = 1, \ldots, \lambda)$, O_i can be constructed from $\Delta_1, \Delta_2, \ldots, \Delta_i$.

4. For any ρ_i $(i = 1, \ldots, \lambda)$, $L_\infty(O_i) \leq \epsilon_i$ and $L_\infty(O_i \setminus \{r_{id,t}\}) > \epsilon_i$ for $\forall r_{id,t} \in O_i$

Conditions (1)–(3) ensure that the set received by the client contains all desired readings and each reading is transmitted exactly once. Condition (4) guarantees that the error bound of each sub-query is satisfied, and only a *just-sufficient reading set* is transmitted to the client for each sub-query ρ_i, i.e., its error bound requirement cannot be satisfied if any reading is removed from O_i.

In HybridDB, each sensor mote stores all its readings locally on high-capacity flash memories for energy-efficiency. Due to the large number of readings on each mote, an index structure is required in order to process each sub-query efficiently. In addition, since the number of index entries is also large and the RAM resource on current sensor platforms is very limited, the index structure must be stored on flash memories as well.

5.3.2 Research Objectives

Our research objectives in this part are three-fold. Firstly, we want to design an efficient data storage system for resource-constrained sensor platforms to handle large-scale datasets. Secondly, we aim to design an efficient query processing mechanism to support incremental ϵ-approximate querying. Finally, we must evaluate the performance of HybridDB in real-world deployments.

5.4 Design Considerations

In this section, we first discuss various factors that make the design of HybridDB challenging. Then we discuss our design principles.

5.4.1 Design Challenges

5.4.1.1 Flash Constraints

Flash memory complicates the design of HybridDB by prohibiting in-place updates. Unlike magnetic disks, flash memories only allow bits to be programmed from 1 to 0. To reset a bit to 1, a large block of consecutive bytes must be erased, which is typically several kilobytes large [78]. There are two kinds of flash memories. NOR flash memories are byte-addressable and permit random access I/O, but their erase blocks are very large. NAND flash memories are page-oriented and limited to sequential writes within an erase block that can be significantly smaller than a NOR flash block. Reads and writes on NAND flash happen at a page granularity. Since each page can be written only once after each complete block erasure, out-of-place writes to an existing NAND page are complex and very expensive. Portable flash packages such as SD cards and CF cards exploit a Flash Translation Layer (FTL) to hide many of these complexities and provide a disk-like interface. However, random page writes on current FTL-equipped devices are still *well over two orders of magnitude more expensive* than sequential writes, while semi-random writes are very efficient [13].

5.4.1.2 Energy Constraints

NOR flash memories and NAND flash memories are very different in speed and energy-efficiency. Table 5.1 shows the latency and energy consumption of each operation on the 512KB Atmel AT45DB041B NOR flash [10, 79] equipped on the Mica family, and the 128MB Toshiba TC58DVG02A1FT00 NAND flash [66] used extensively in the research community. Each NAND block consists of 32 pages of 512B each. We can observe that the NAND flash has a much larger storage capacity, and much faster and more energy-efficient I/O, while the only advantage of the NOR flash is random access and byte-addressable. These features influence the design of HybridDB extensively.

Table 5.1: Performance of flash memory operations

	Atmel NOR (per byte)		Toshiba NAND (per page)	
	Latency	Energy	Latency	Energy
Read	$12.12\mu s$	$0.26\mu J$	$969.61\mu s$	$57.83\mu J$
Write	$12.6\mu s$	$4.3\mu J$	$1081.42\mu s$	$73.79\mu J$
Block Erase	$12ms/2KB$	$648\mu J/2KB$	$2.6ms/16KB$	$65.54\mu J/16KB$

5.4.1.3 Memory Constraints

RAM is very limited on sensor platforms. Current low-end sensor platforms (e.g., MicaZ, Iris and Tmote Sky) are equipped with no more than 10KB RAM. Even on advanced sensor platforms (e.g., iMote2) with tens of megabytes RAM, RAM is still a very precious resource, because complex data processing applications with much higher RAM demands are expected to run on these platforms. Therefore, HybridDB must be designed to minimize the RAM footprint.

5.4.1.4 Incremental Set Computation

The key issue in incremental ϵ-approximate querying is how to compute the incremental set Δ_i ($i = 1, 2, \ldots, \lambda$) efficiently. Due to the limited RAM resource and potentially large cardinality of Ψ_i, it is impossible for a sensor mote to maintain a data structure in RAM to mark which readings have been already retrieved in previous sub-queries. Although these information can be stored in flash memories, it will be very expensive to update them and complicate the storage management of flash memories. Therefore, we must develop a mechanism that can compute Δ_i for each sub-query ρ_i efficiently *on the fly* with little overhead.

5.4.2 Design Principles

Given the above challenges, the design of HybridDB should follow a few design principles. Firstly, the system should take advantage of both the on-board NOR flash and external NAND flash. To support both raw NAND flash and FTL-equipped devices, random page writes should be avoided. To increase the energy-efficiency and storage-efficiency, out-of-place writes to an existing NAND page should be eliminated as well.

Secondly, writes should be batched to match the write granularity of the NAND flash, which can be satisfied by using a page write buffer in RAM. In addition, since the NAND flash is much faster and more energy-efficient, most or even all reads should happen in the NAND flash.

Thirdly, the system should support multiple storage allocation units and align them to erase block boundaries to minimize reclamation costs. Moreover, the system should maintain most data structures and information in flash memories whenever possible to minimize the RAM footprint.

Fourthly, HybridDB should support data aging to reclaim space for new data when the NAND flash starts filling up with minimum overhead. Finally, each sensor mote should organize its readings in such a way that readings in Ψ_i can be efficiently located with the help of the index structure, and Δ_i can be computed efficiently for each sub-query ρ_i.

Chapter 6

HybridStore: An Efficient Flash-based Data Management System

In this chapter, we present the design details of HybridStore, which is a novel efficient resource-aware data storage system used by HybridDB to store and query readings *in situ* on each sensor mote, following the design principles discussed in Chapter 5. Note that HybridStore can be used independently as a general data storage system as [1, 9]. Therefore, instead of describing how to process a sub-query ρ_i, we design HybridStore to process more general queries with only time windows and value ranges, but no error bounds, as the selection predicates. In Chapter 7, we will explain how to modify HybridStore slightly for ρ_i to compute Δ_i.

HybridStore provides the following interface to insert new sensor readings and query existing readings:

- command error_t insert(float *key*, void* *record*, uint8_t *length*)
- command error_t select(uint32_t t_1, uint32_t t_2, float k_1, float k_2)

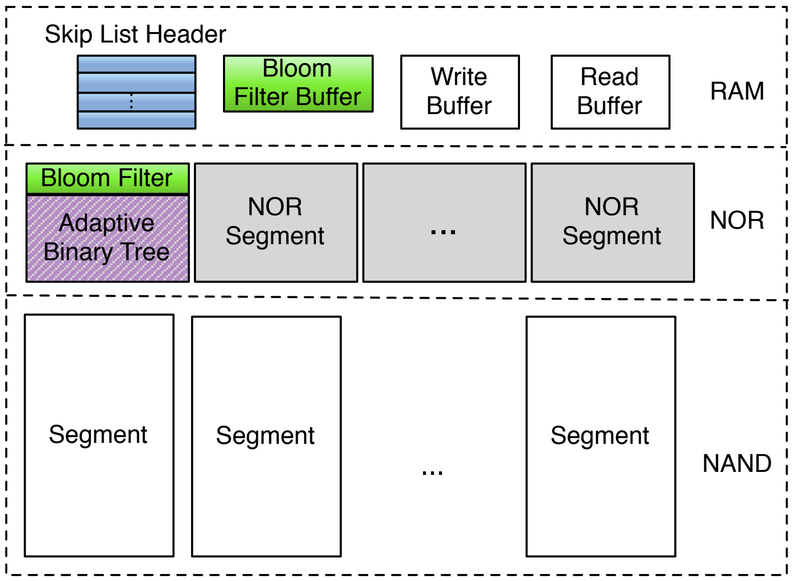
The `insert` function inserts a reading $r_{id,t}$ (excluding the field *id*) to the storage system and updates the index structure, while the `select` function supports joint queries involving both time windows ($[t_1, t_2]$) and key ranges ($[k_1, k_2]$) as their *selection predicate*. HybridStore consists of the following main components: Storage Manager, Index Manager, Query Processor, Data Aging and Space Reclamation Module, and Failure Recovery Module.

6.1 Storage Manager

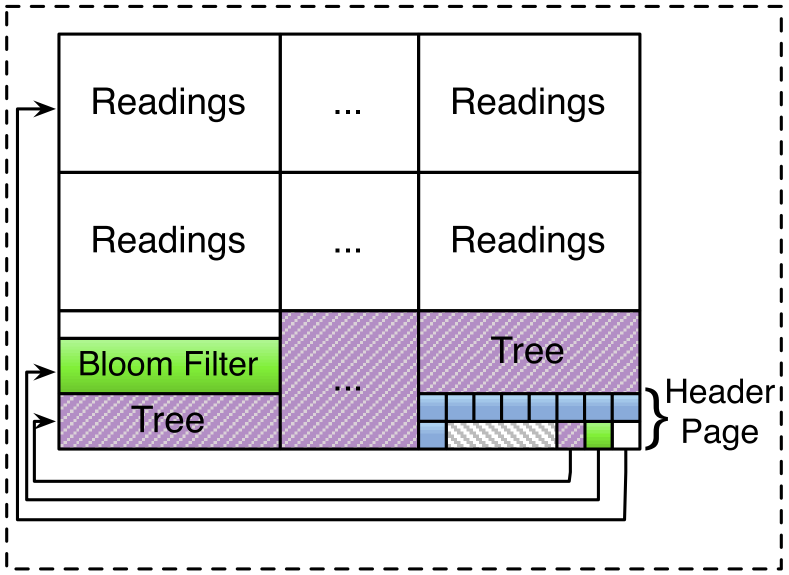
The Storage Manager allocates storage space from the NOR flash and the NAND flash for index construction and data storage upon request. Fig. 6.1a shows the storage hierarchy of the system. Both the NOR flash and the NAND flash are organized as circular arrays, resulting in the minimum RAM overhead, because we do not need to maintain a data structure in RAM to track free blocks. In addition, this organization directly addresses the write constraints, space reclamation, and wear-leveling requirements (Section 6.4).

At the highest level, the NOR flash is divided into *equally-sized* segments, each of which consists of one or multiple consecutive erase blocks. Storage is allocated and reclaimed at the granularity of a segment. The NAND flash is allocated at the granularity of an erase block, but reclaimed at the granularity of a segment that *logically* consists of several consecutive erase blocks storing readings, the index of this segment copied from the corresponding NOR segment (colored in green and purple), and the header page, as shown in Fig. 6.1b.

Only four absolutely necessary data structures are maintained in RAM. The write buffer is of one page size to batch the writes to the NAND flash, and the read buffer is two pages large (one for index page reads and the other for data page reads). The other two data structures are the skip list header and Bloom filter buffer that are discussed in the next section. Our design not only minimizes the RAM footprint of HybridStore and complies with the write and read granularity of NAND flash, but also makes the failure recovery mechanism simple and efficient (Section 6.5).



(a) Storage hierarchy



(b) NAND segment structure

Figure 6.1: System architecture

6.2 Index Manager

In this section, we present the most important component of HybridStore: the *Index Manager*. HybridStore leverages a memory hierarchy to achieve more efficient index operations. Specifically, HybridStore divides a sensor data stream into dynamically-sized partitions, each of which is stored in a logical NAND segment. The size of a stream partition is decided by the size of a NOR segment, and the distribution of the key values of readings in that partition. The index for this partition is first "cached" in a NOR segment, which is then copied to the corresponding logical NAND segment when it is filled. Next, HybridStore allocates a new NOR segment for the next partition, and stores its readings from the *next page* on the NAND flash memory.

HybridStore exploits an *inter-segment skip list* to locate the segments covered by $[t_1, t_2]$ efficiently. Within each segment, HybridStore maintains an *in-segment* β*-Tree* to locate all readings within $[k_1, k_2]$ efficiently. To speed up the processing of value-equality queries (i.e., $k_1 = k_2$), an *in-segment Bloom filter* is created for each segment to quickly detect the existence of a given key.

HybridStore chooses the partition-based index scheme instead of a global index as in [1, 9] for the following reasons. Firstly, typical queries on sensor data always involve time windows. Especially, readings in small time windows are more interesting due to their temporal correlations. Since each logical NAND segment only stores the readings of a partition that corresponds to a small time window, many logical NAND segments outside the query time window can be skipped, reducing a substantial number of unnecessary page reads during query processing. Secondly, the number of readings in a partition is very limited compared to that in the whole steam. This allows index structure optimization and much cheaper index construction costs. Thirdly, the range of the key values of readings in a partition is very small compared to the whole range of all possible key values. When processing a query with a value range within its selection predicate, many logical NAND segments outside the query value range can be skipped as well, further reducing many unnecessary page reads. Therefore, HybridStore is extremely efficient to process joint queries with both time windows and value ranges as their selection predicates. Finally, since all logical NAND segments are relatively independent of each other, HybridStore can support time-based data aging without any garbage collection mechanism, resulting in the substantially reduced overhead.

Now we discuss the index structure of HybridStore in details, which consists of three main modules: the inter-segment skip list, the in-segment β-Tree, and the in-segment Bloom filter. In the last subsection, the procedure to copy the "cached" index from the NOR flash to the NAND flash is described as well.

6.2.1 Inter-segment Skip List

The key issue to process a query with a time window is to locate the segments containing readings within that time window. A naive approach is to scan the headers of all the segments one by one, assuming the time window of all the readings in a segment is available in its header and all segments are chained using previous segment address pointers. The expected number of page reads to locate the most recent segment within the query time window is linear with the number of segments. However, we actually can do much better by exploiting the fact that the time windows of all segments are naturally ordered in descending order. Note that the desired segments cannot be located using binary search on timestamps, because neither the size of a segment nor the number of readings in that segment is fixed.

To facilitate efficiently locating the desired segments for a query, we organize all segments as a skip list [80]. A skip list is an ordered linked list with additional forward links added randomly, so that a search in the list can quickly skip parts of the list. The expected cost for most operations is $O(\log n)$, where n is the number of items in the list. Since segments are created in increasing order of timestamp, a new segment is always inserted at the front of the skip list, which can be efficiently implemented in a flash.

The inter-segment skip list consists of a header node in RAM and a node in the header page of each segment (colored in blue in Fig. 6.1). Each node keeps H forward pointers, each of which references the address of the header page of a segment and the timestamp of the first (oldest) reading stored in that segment. All pointers in the header node are initialized to *null* at first. When a segment is full, the skip-list node in its header page is created and inserted into the skip list before a new segment starts as follows. Firstly, a level $h \in [1, H]$ is generated for it randomly, such that a fraction q ($q = \frac{1}{2}$ typically) of the nodes with level j can appear in level $j + 1$. The maximum level of all segments in the current system is updated if it is smaller than h. Then every pointer in level $j \in [1, h]$ in the skip-list header is copied as the level j pointer to the header page. Finally, the timestamp of the first reading in this segment and the header page address are written as the new level j pointer to the skip-list header.

The header page of each segment contains the timestamps of the first and the last readings stored in this segment. To search the segments containing readings within $[t_1, t_2]$, we first locate the most recent segment with a *start* timestamp smaller than t_2, using an algorithm similar to the search algorithm in [80]. The subsequent segments can be located by following the level 1 pointer in the skip-list node of each segment, until a segment with a start timestamp smaller than t_1 is encountered.

6.2.2 In-segment β-Tree

To support value-based equality and range queries, HybridStore exploits an adaptive binary tree structure, called β-Tree, to store the index for each segment. The β-Tree consists of a set of equal-sized buckets, each of which stores index entries within a certain value range, while the root bucket is with range $[-\infty, \infty]$. The header of a bucket consists of its value range, the bucket IDs of its left and right child, and the value to split its value range to obtain the value ranges for its children. The β-Tree is first created and updated in a NOR segment, and then copied to the corresponding logical NAND segment when this NOR segment is full.

An index entry $< key, addr >$ is inserted into the β-Tree as follows. Suppose the current bucket is b and $key \in (b.min, b.max]$, then this entry is appended to b. Otherwise, we traverse the β-Tree from the root bucket to locate the leaf bucket b' such that $key \in (b'.min, b'.max]$, and append this entry to b'. If b (or b') is full, its value range is split into $(b.min, mid]$ and $(mid, b.max]$, and a new bucket b_{new} is allocated as its left child if $key \leq mid$, or as its right child otherwise. Then the headers of both b (or b') and b_{new} are updated correspondingly and this entry is appended into b_{new}. Since the children of a bucket are allocated only if necessary, it is possible to have $b' = null$ in the above case if, for example, b' is the left child of its parent but only the right child of its parent has been allocated since its splitting. In this case, a new bucket is allocated for b' first.

Instead of splitting the value range of a bucket evenly as in [1,9], HybridStore exploits a prediction-based adaptive bucket splitting method, because readings are temporally correlated, which can be used to predict the value range of the following readings based on the most recent readings, and split a bucket range accordingly. This method is preferred for the following reasons. Firstly, each partition corresponds to a small time window, and thus contains readings in a small value range. If the very large range for all possible key values is split evenly in each step, the index tree of a segment may degenerate to a long list at the beginning, resulting in more time and energy consumption to traverse the index. Secondly, although the whole range is very large, most readings will belong to a much smaller range due to their uneven distribution. As shown in Fig. 11 in [9], over 95% of the temperature measurements belong to $[30\,°\mathrm{F}, 80\,°\mathrm{F}]$, while the whole range is $[-60\,°\mathrm{F}, 120\,°\mathrm{F}]$. Again, the evenly splitting method will result in a rather unbalanced tree.

HybridStore exploits the Simple Linear Regression estimator for prediction due to its simplicity in computation, negligible constant RAM overhead, and high accuracy for temporally correlated data. HybridStore buffers the *keys* of the most recent n_{idx} readings and predicts the value range for the following $2n_{idx}$ readings, where n_{idx} is the number of entries that can be stored in a bucket. Suppose the range of the current bucket is $(b.min, b.max]$ and the predicted range is $[x, y]$, the splitting point *mid* is computed as:

$$mid = \begin{cases} \frac{x+y}{2} & [x, y] \subseteq (b.min, b.max] \\ \frac{b.min+b.max}{2} & (b.min, b.max] \subseteq [x, y] \\ \frac{b.min+y}{2} & x \leq b.min < y \leq b.max \text{ and } \frac{2n_{idx}*(y-b.min)}{y-x} > n_{idx} \\ \max(y, \frac{b.min+b.max}{2}) & x \leq b.min < y \leq b.max \text{ and } \frac{2n_{idx}*(y-b.min)}{y-x} \leq n_{idx} \\ \frac{x+b.max}{2} & b.min \leq x < b.max \leq y \text{ and } \frac{2n_{idx}*(b.max-x)}{y-x} > n_{idx} \\ \min(x, \frac{b.min+b.max}{2}) & b.min \leq x < b.max \leq y \text{ and } \frac{2n_{idx}*(b.max-x)}{y-x} \leq n_{idx} \end{cases}$$

The intuition is that if among the next $2n_{idx}$ readings, the expected number of readings belong to $(b.min, b.max]$ is more than n_{idx}, the value range should be split such that these readings can be distributed evenly to the left and right child buckets. Otherwise, the value range should be split such that all these readings can be stored in the left or right child bucket alone, while avoiding splitting the value range too unevenly whenever possible.

Compared to MaxHeap [1], which is an evenly splitting scheme, HybridStore can generate a more balanced tree. For example, suppose each bucket can store the index entries generated in half an hour, the current temperature is 80 °F and will increase 1 °F every half an hour, and the whole range is $[-60\,°\text{F}, 120\,°\text{F}]$. Assuming HybridStore can predict accurately, the root will be split with $mid = 82\,°\text{F}$, and its right child will be split with $mid = 84\,°\text{F}$. The resulting β-tree will have three layers for readings in the following 2.5 hours, while MaxHeap degenerates to a list with 5 buckets. In addition, MaxHeap allocates two child buckets at the same time when a bucket needs to be split, resulting in more wasted space with empty buckets. To handle the uneven distribution of key values, MaxHeap selects a bucket for an index entry based on the hashed key value, but stores the unhashed key. As a result, MaxHeap can only retrieve *discrete values* in a range search. More importantly, however, any spatial correlation of index insertions is destroyed. After hashing, the index entries for consecutive readings are very likely to be stored in many different buckets, which will increase the read costs both for bucket locating and query processing substantially. On the contrary, these entries will be stored in the same bucket in the β-Tree. Finally, different from [1, 9], our prediction-based adaptive splitting scheme does not require a priori knowledge of the whole key value range, which makes HybridStore more suitable for general applications.

As the β-Tree grows, the value range of each bucket becomes smaller and smaller. Due to the vibration of key values, a sequence of consecutive index entries may need to be appended to a few nearby buckets back and forth. The above design requires HybridStore to traverse the β-Tree from the root all the way to the desired leaf bucket whenever a bucket switch is needed. To improve the performance and avoid traversing the β-Tree to switch between a few nearby buckets back and forth, HybridStore exploits a small least-recently-used cache to store the value ranges and IDs for a few buckets. Section 6.6.1 shows that the average insertion latency can be significantly reduced with an information cache for only 5 buckets.

6.2.3 In-segment Bloom Filter

A special case of value-based queries is value-based *equality* search that is also often desired [1]. Although β-Trees can support this kind of queries, HybridStore needs to traverse the whole β-Tree even when the given key does not exist in a segment. To better support these queries, HybridStore creates a Bloom filter [81] for each segment to detect the existence of a key value in this segment efficiently.

A Bloom Filter (BF) is a space-efficient probabilistic data structure for membership queries in a set with low false positive rate but no false negative. It uses a vector of c bits (initially all set to 1) to represent a set of n_{bf} elements, and f independent hash functions, each producing an integer $\in [0, c-1]$. To insert an element a, the bits at positions $h_1(a), \ldots, h_f(a)$ in the bit vector are cleared to 0. Given a query for element a', all bits at positions $h_1(a'), \ldots, h_f(a')$ are checked. If any of them is 1, a' cannot exist in this set. Otherwise we assume that a' is in this set.

Since a Bloom filter requires bit-level random writes, it must be buffered in RAM. However, if a single Bloom filter is used to represent all readings in a segment, this buffer size may be very large in order to keep $p_{\mp}$ very low. For example, suppose a segment can store 4096 readings and three hash functions are used, in order to keep $p_{\mp} \approx 3.06\%$, then the size of its Bloom filter buffer must be at least 4KB.

To reduce the RAM footprint, HybridStore *horizontally* partitions the big Bloom filter of a segment into a sequence of small fix-sized Bloom filters sections, and allocates a small buffer in RAM for a section. Suppose the buffer size is c

bits, the number of hash functions is f, and the desired false positive rate is $p_{\mp}$, the maximum number n_{bf} of readings that a BF section is able to represent can be calculated from the equation $p_{\mp} = \left(1 - \left(1 - \frac{1}{c}\right)^{n_{bf}f}\right)^{f}$. Whenever n_{bf} readings have been inserted into the current BF section, the BF buffer is flushed to the current NOR segment, and then initialized for the next section. In our implementation, $c = 2048$ bits, $f = 3$, $p_{\mp} = 3.06\%$, and $n_{bf} = 256$.

Algorithm 1 checkBF($addr, l_{frag}, key$)

```
Input: $addr$: start address of the BF pages, $l_{frag}$: length of a BF fragment in bytes,
    $key$: key value
Output: $true$ if there is a record with the given key in this segment; $false$ otherwise
 1: $code \leftarrow$ hashcode($key$); $bv \leftarrow$ createBitVector($\lceil \frac{B_{page}}{l_{frag}} \rceil$);
 2: for $i = 0 \rightarrow code.size$ do
 3:     $bfPage \leftarrow$ loadPage($addr + \lfloor \frac{code[i]}{8l_{frag}} \rfloor * B_{page}$); $bv.setAll()$;
 4:     for $j = 0 \rightarrow bv.size$ do
 5:         $mask \leftarrow 0x80 >> (code[i] \ \% \ 8)$; $offset \leftarrow code[i] \ \% \ 8l_{frag}$;
 6:         if $bfPage[j * l_{frag} + \lfloor \frac{offset}{8} \rfloor] \ \& \ mask \neq 0$ then $bv.clear(j)$; end if
 7:     end for
 8:     $exist \leftarrow false$;
 9:     for $i = 0 \rightarrow bv.size$ do
10:         $exist = exist \mid bv.get(i)$;
11:     end for
12:     if $!exist$ then return $false$; end if
13: end for
14: return $true$;
```

A drawback of horizontal partitions is that all BF sections of a segment must be scanned to decide whether the given key exists in this segment. HybridStore addresses this drawback by *vertically* splitting these BF sections into fragments and grouping them into pages when the NOR segment is copied to the logical NAND segment. Assume there are s BF sections in the current segment when it is full. Then the size of a fragment is $l_{frag} = \left\lfloor \frac{B_{page}}{s} \right\rfloor$ bytes, so that the bits in the range $[i * 8l_{frag}, (i+1) * 8l_{frag} - 1]$ from every BF section are grouped to page $i \in \left[0, \lceil \frac{B_{page}}{l_{frag}} \rceil - 1\right]$. Here, B_{page} is the size of a NAND flash page in bytes. Thus HybridStore only needs to scan at most f pages at $\left\lfloor \frac{h_1(key)}{8l_{frag}} \right\rfloor, \ldots, \lfloor \frac{h_f(key)}{8l_{frag}} \rfloor$ when checking a key value key, as shown in Algorithm 1. For each hash code $code[i]$, HybridStore firstly loads the page containing all the $code[i]$-th bits of every BF section (Line 3), and then check the corresponding bit in each BF fragment (Line 4–7). If the corresponding bit is not set in any fragment in that page (Line 9–11), we can conclude that this key does not exist (Line 12).

Note that the maximum number of readings in a segment must be no more than $n_{bf} * B_{page}$ to guarantee $l_{frag} \neq 0$, which is a valid assumption. For example, in our implementation, $B_{page} = 512$, $n_{bf} = 256$ and the storage overhead to index each reading is at least 9 bytes (4 bytes for the key, 4 bytes for the address pointer and at least 1 byte for the Bloom filter). Therefore, the maximum size of a NOR segment is at least 1.125MB, which is even larger than the capacity of NOR flash on most current sensor platforms.

6.2.4 Copy Index from the NOR flash to the NAND flash

Within a NOR segment, BF sections are stored sequentially from the beginning, while β-Tree buckets are stored sequentially from the end. Since the NAND flash is much faster and more energy-efficient, the index of a segment is copied to the NAND flash after the NOR segment is full as follows. Firstly, the BF sections are copied as described above. Secondly, the β-Tree is copied and multiple consecutive buckets are written to the same page if they can fit in. The Query Processor is able to translate a bucket ID to the right page address and offset to load the desired bucket. Hence, the bucket size should be $\frac{B_{page}}{2^i}$ (256 bytes is recommended for the sake of storage-efficiency). Then the current NOR segment is erased. Finally, the time window and value range of all readings in this segment, the addresses of the first page for readings, for Bloom filter and for the β-Tree, the length of a BF fragment, and the skip-list node are written to the *next page*, which is the header page of this segment. Thus, all page writes in the NAND flash are purely sequential; the reason why HybridStore can support both raw NAND flash chips and FTL-equipped NAND flash cards efficiently.

6.3 Query Processor

In this section, we show how HybridStore can efficiently process joint queries involving both time windows and value ranges as their selection predicates, as shown in Algorithm 2. The basic idea is to skip all the segments that do not satisfy the selection predicate by checking their header pages, or do not contain the given key by checking their Bloom filters. For the current segment, into which new readings are inserting, its index is still maintained in the corresponding NOR segment. However, the process is very similar, except that the access to its index is redirected to the NOR segment, and the information to be loaded from the header page are available in RAM.

HybridStore starts by locating the most recent segment within the time window using the inter-segment skip list (Line 1), and then scans segments sequen-

Algorithm 2 select(t_1, t_2, k_1, k_2)

```
Input: Time window $[t_1, t_2]$ and key value range $[k_1, k_2]$ of a query
Output: The readings that satisfy the query criteria
 1: addr $\leftarrow$ skipListSearch($t_2$);
 2: while $addr \geq 0$ do
 3:   addr $\leftarrow$ segmentSearch($addr, t_1, t_2, k_1, k_2$);
 4: end while
 5: signal $finished$;

 6: function SEGMENTSEARCH($addr, t_1, t_2, k_1, k_2$)
 7:   $hp \leftarrow$ loadPage($addr$);                              ▷ Load the header page
 8:   if $[k_1, k_2] \cap [hp.minK, hp.maxK] \neq \emptyset$ then
 9:     if $(t_1 == t_2)$ || $([hp.minK, hp.maxK] \subseteq [k_1, k_2])$ then
10:       directRetrieve($hp, t_1, t_2$);
11:       return $(hp.startT > t_1 \wedge hp.sl[0].time \geq Sys.minT)$ ? $hp.sl[0].addr : -1$;
12:     else if $k_1 == k_2$ then                              ▷ Value-based equality query
13:       if checkBF($hp.bfAddr, hp.bfFragSize, k_1$) == $false$ then
14:         return $(hp.startT > t_1 \wedge hp.sl[0].time \geq Sys.minT)$ ? $hp.sl[0].addr : -1$;
15:       end if
16:     end if
17:     $queue \leftarrow$ createQueue($hp.idxAddr$);                        ▷ Traverse $\beta$-tree
18:     while !$queue.empty()$ do
19:       $b \leftarrow$ loadBucket($queue.dequeue()$);
20:       for $i = 0 \rightarrow b.entries.size$ do
21:         if $(b.entries[i] == null)$ || $(b.entries[i].addr \geq hp.bfAddr)$ then break;
   end if
22:         if $b.entries[i].key \in [k_1, k_2]$ then
23:           $dataP \leftarrow$ loadPage($b.entries[i].addr$);
24:           if $dataP[b.entries[i].addr \% B_{page}].timestamp \in [t_1, t_2]$ then
25:             signal $dataP[b.entries[i].addr \% B_{page}]$;
26:           end if
27:         end if
28:       end for
29:       if $(b.left \neq null) \wedge (b.middle \geq k_1)$ then $queue.enqueue(b.left)$; end if
30:       if $(b.right \neq null) \wedge (b.middle < k_2)$ then $queue.enqueue(b.right)$; end if
31:     end while
32:   end if
33:   return $(hp.startT > t_1 \wedge hp.sl[0].time \geq Sys.minT)$ ? $hp.sl[0].addr : -1$;
34: end function
```

tially until the whole time window has been covered (Line 2–4). For each segment, its header page is loaded first. If this segment potentially contains readings within the value range of the query (Line 8), HybridStore continues to retrieve the readings. Otherwise, this segment will be skipped. Two special cases are treated separately. Firstly, if the value range of this segment is completely contained by the query value range (Line 9), HybridStore can retrieve the desired readings directly, without traversing the β-Tree to locate them. The `directRetrieve` function first computes the address of the first reading in $[t_1, t_2]$ as $hp.dataAddr + \max\{0, \lfloor \frac{t_1 - hp.startT}{\tau} \rfloor \times B_{rec}\}$, where B_{rec} is the size of a sensor reading in bytes. Then it retrieves readings sequentially from there until a reading outside $[t_1, t_2]$ is encountered, or the end of this segment is reached. Time-based *equality* queries (Line 9) are processed in the same way as a special case. Secondly, for value-based *equality* queries (Line 12), HybridStore first checks the existence of the key in this segment using Algorithm 1. If this key does not exist, this segment will be skipped as well. For general joint queries, the β-Tree of this segment is traversed using the Breadth-First Search algorithm (Line 17–31). An index entry may become invalid due to a sudden power failure (Section 6.5), because the last a few readings that are stored in the write buffer right before the power failure are lost, but their index entries are inserted into the β-Tree successfully. After failure recovery, the page that is supposed to store the lost readings will become the first page for Bloom filter. These invalid index entries can be recognized by checking the second condition in Line 21. Note that HybridStore can be easily extended to support traditional aggregation queries (e.g., MAX, MIN, AVERAGE, etc.), which is beyond the focus of this work.

To reduce the RAM footprint, HybridStore returns readings on a record-by-record basis. Actually, the β-Tree traversal is also implemented in a split-phase fashion (bucket-by-bucket using the *signal-post* mechanism), although a queue-based implementation is presented here for clarity. In addition, to take advantage of the temporal correlations and spatial locality of readings, a small address pool is applied here to buffer the addresses of the readings to be loaded. Therefore, instead of loading a data page immediately as in Line 23, HybridStore first scans the address pool to remove any address that is smaller than the new reading address *addr*, but belongs to the same data page. Then *addr* is added to the pool if it is not full. Otherwise, HybridStore removes one address from the pool and loads its corresponding data page. If this address points to the last reading that satisfies the selection predicates in that data page, all readings satisfying the selection predicates in this page will be returned. Otherwise, they will be ignored. After that, *addr* is added to the pool. When the β-Tree traversal is finished, the pool is cleared up by removing addresses from it and loading the corresponding data pages iteratively. Obviously, each desired reading will be returned exactly once, even though its data page may be loaded multiple times.

6.4 Data Aging and Space Reclamation

As shown in [10], a sensor mote can store over 10GB data during its lifetime. If the capacity of the external NAND flash is not big enough to store all these data, some data need to be deleted to make room for future data as the flash starts filling up. HybridStore exploits a simple time-based data aging mechanism to discard the oldest data. When no space is available on the NAND flash to insert the current reading, HybridStore will locate the oldest segment and erase all the blocks in that segment, except the last block if its header page is not the last page in this block. Then the minimum timestamp of all the readings currently stored in the system (i.e., `Sys.minT`) is updated. Since NAND flash is organized as a circular array, wear-leveling is trivially guaranteed. In addition, since segments are independent of each other, no garbage collection mechanism is needed here. On the contrary, other index schemes (e.g., [1,66,71]), require extensive page reads and writes to move valid pages within the reclaimed erase blocks to new locations, and maintain extra data structures in flash or RAM.

Note that we do not need to delete the pointers referencing the reclaimed segment from the skip list, even though they become invalid now. This problem is handled by the `select` algorithm (Line 11, 14 and 33). Whenever a pointer with a timestamp smaller than `Sys.minT` is encountered, the `select` algorithm knows that this pointer is invalid and the query processing is already completed successfully. On the contrary, for *each invalid index entry*, MicroHash [9] must load the referenced data page to learn that this page has been deleted and re-used, resulting in many unnecessary page reads.

6.5 Failure Recovery

A failure recovery mechanism that can cope with sudden power failures is required for a resilient storage system. Existing storage systems, such as [9, 12], rely on checkpointing, in which data structures maintained in RAM are periodically written into a special checkpoint area in the NAND flash. Checkpointing complicates the management of the NAND flash and violates the wear-leveling requirement of NAND pages, while providing only a coarse level of data consistency (modifications after the last checkpoint are not guaranteed to be recovered). HybridStore exploits a simple and efficient recovery mechanism that takes advantage of only existing data structures stored on flash memories without the need for any checkpoint, and provides the highest level of data consistency.

6.5.1 Recovery Algorithm

To recover from a failure, the following information must be restored: the information to be written to the header page of the latest segment, the Bloom filter buffer and the skip-list header node. Firstly, HybridStore scans the NOR flash by reading an index entry idx at $B_{seg} - B_{bkt}$ that is supposed to be the in-segment offset of the first index entry from each NOR segment (remind that β-Tree buckets are stored from the end in each NOR segment). Here, B_{seg} and B_{bkt} are the sizes of a NOR segment and a β-Tree bucket in bytes, respectively. The NOR segment with any byte in idx not equal to $0xFF$ is the one storing the index structure for the last stream partition, and $idx.addr$ points to the NAND page storing the first reading in this partition. By loading this page, the lower bound of the time window of this segment can be recovered.

Secondly, HybridStore scans from the beginning of this NOR segment to find out the number of valid BF sections. It reads from each potential section byte by byte. If any byte is not equal to $0xFF$, it increases the counter by one and jumps to the next potential section. This process repeats until a section with *all* bytes equal to $0xFF$ is encountered. Suppose there are η valid BF sections. Then HybridStore begins to scan the NAND flash from $idx.addr + n_{bf} * \eta * B_{rec}$ to retrieve all valid readings sequentially to restore the Bloom filter buffer and the upper bound of the time window of this segment. Next, the Bloom filter buffer is flushed to the last BF section with all bytes equal to $0xFF$ in the NOR segment, whose content is then copied to the corresponding NAND segment as specified in Section 6.2.4. During this process, the value range of this segment, the addresses of the first page for the Bloom filter and for β-Tree, and the length of a BF fragment can be recovered.

Finally, the skip-list header node in RAM can be restored as follows. Firstly, HybridStore loads the header page hp_1 of the last segment that has been saved successfully at $idx.addr - B_{page}$. Suppose the maximum level of its skip-list node is h_1. Then the tuple $\langle idx.addr - B_{page}, hp_1.startT \rangle$ is written as the level j pointer to the skip-list header for $\forall j \in [1, h_1]$. Next, HybridStore follows the level h_1 pointer to load the header page hp_2 of an older segment at $hp_1.sl[h_1].addr$. Suppose the maximum level of the skip-list node in this segment is h_2. Obviously, $h_2 \geq h_1$. If $h_2 > h_1$, the tuple $\langle hp_1.sl[h_1].addr, hp_2.startT \rangle$ is written as the level j pointer to the skip-list header for $\forall j \in [h_1 + 1, h_2]$. Otherwise, HybridStore just continues to load the header page at $hp_2.sl[h_2].addr$. This process repeats until an invalid pointer or a skip-list node with a *null* pointer in every level from h_1 to H is encountered, or H pointers have been restored to the skip-list header node, as shown in Fig. 6.2.

After that, the latest segment can be inserted into the skip list as described in Section 6.2.1, the header page can be written to the *next page* in the latest NAND segment, and a new stream partition starts.

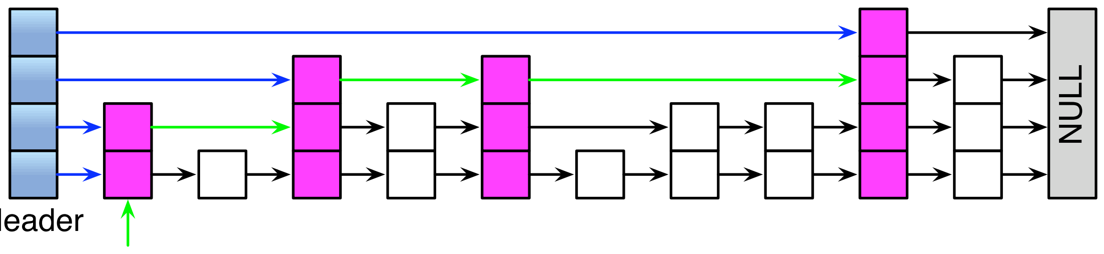


Figure 6.2: Skip-list header recovery ($H = 4$): HybridStore loads the magenta skip-list nodes from their corresponding header pages following the green pointers; The restored skip-list header consists of all blue pointers.

Note that only readings stored in the NAND write buffer right before the power failure are lost, which cannot be recovered by any failure recovery mechanism as long as a write buffer is used. Therefore, HybridStore provides the highest level of data consistency. However, their index entries are inserted successfully. This problem can be solved by Line 21 in Algorithm 2, which does not induce any NAND page read overhead.

In addition, the value of `Sys.minT` is not restored by our recovery mechanism, which is used only when the Query Processor is processing a query with t_1 smaller than `Sys.minT` (Line 11, 14 and 33 in Algorithm 2). HybridStore exploits a passive approach to recover `Sys.minT` to reduce the recovery overhead. `Sys.minT` is initiated to 0 upon a failure recovery, and then updated in the following two available ways. Firstly, if the data aging function is triggered to delete the oldest segment, it can be recovered immediately from the header page of the new oldest segment. Secondly, during the processing of the *first* query with t_1 smaller than the *actual* value of `Sys.minT` in Algorithm 2 after the recovery, the Query Processor will encounter a segment with invalid pointers in the header page when it tries to load a data page, or a β-Tree bucket. Then HybridStore will load the *next* page following this header page, which is the first data page of the oldest segment in the current system. `Sys.minT` is restored as the timestamp of the first reading in this data page.

6.5.2 Expected Overhead

In this section, we analyze the expected overhead in terms of latency and energy consumption for HybridStore to recover from a failure. We denote the cost to read a *byte* from the NOR flash as C_{NOR}, and the cost to read a *page* from the NAND flash as C_{NAND}, respectively. Firstly, the expected cost to locate the NOR segment for the latest stream partition and the lower bound of the time window is:

$$\frac{B_{NOR}}{2 \times B_{seg}} \times B_{idx} \times C_{NOR} + C_{NAND} \tag{6.1}$$

where B_{NOR} is the capacity of NOR flash, and B_{idx} is the size of an index entry in bytes, respectively.

Secondly, in each BF section in the NOR segment located above, the probability that a certain bit is set is $p_{set} = 1 - \left(1 - \frac{1}{c}\right)^{n_{bf}f}$. Therefore, the expected number of bytes that will be read from each valid BF section by the recovery algorithm is $\sum_{j=1}^{\frac{c}{8}-1} j p_{set}^{8(j-1)}(1 - p_{set}^{8}) + \frac{c}{8} p_{set}^{c-8} = \frac{1-p_{set}^{c}}{1-p_{set}^{8}}$. Note that all bytes in the last BF section must be read because its content has not been flushed to the NOR segment. Obviously, the maximum number of BF sections in each NOR segment is $\left\lfloor \frac{B_{seg}}{\frac{c}{8}+\frac{B_{bkt}}{n_{idx}}\times n_{bf}} \right\rfloor$. In addition, the expected number of pages that will be read from the NAND flash to reconstruct the BF buffer is $\left\lceil \frac{n_{bf}}{2\times\lfloor\frac{B_{page}}{B_{rec}}\rfloor} \right\rceil$. Therefore, the expect cost to recover the BF buffer and the upper bound of the time window is:

$$\left(\frac{1}{2}\left\lfloor \frac{B_{seg}}{\frac{c}{8}+\frac{B_{bkt}}{n_{idx}}\times n_{bf}} \right\rfloor - 1\right)\times\frac{1-p_{set}^{c}}{1-p_{set}^{8}}\times C_{NOR}+\frac{c}{8}C_{NOR}+\left\lceil \frac{n_{bf}}{2\times\lfloor\frac{B_{page}}{B_{rec}}\rfloor} \right\rceil\times C_{NAND} \tag{6.2}$$

Finally, to analyze the expected cost to recover the skip list header, we model the third step in Section 6.5.1 as an absorbing Markov chain. Specifically, state $X_{h,m}$ $(1 \le h \le H, 1 \le m \le M)$ represents that the header page of the m^{th} segment is loaded, which contains a skip-list node of maximum level h. Here, M is the total number of segments currently stored in HybridStore and the first segment is the oldest one. The absorbing state $\otimes$ represents that the skip list header has been recovered successfully. In a skip list, the probability that a particular node is of level h or higher is $\mathbb{P}\{Y \ge h\} = q^{h-1}$, and of exact level h is:

$$\mathbb{P}\{Y = h\} = \begin{cases} q^{h-1}(1-q) & 1 \le h \le H-1 \\ q^{h-1} & h = H \end{cases}$$

In this Markov chain, a transition from X_{h_1,m_1} to X_{h_2,m_2} $(h_1 < H, h_1 \le h_2, m_1 > m_2)$ means that the level h_1 pointer in the skip-list node of the m_1^{th} segment points to the m_2^{th} segment, which can happen only if the maximum level of the skip-list node in each segment $m_1 - 1, m_1 - 2, \ldots, m_2 + 1$ is smaller than h_1. Therefore, we have:

$$\mathbb{P}\{X_{h_2,m_2} \mid X_{h_1,m_1}\} = (1-q^{h_1-1})^{m_1-m_2-1} \times \mathbb{P}\{Y = h_2\} \quad (h_1 < H, h_1 \le h_2, m_1 > m_2) \tag{6.3}$$

A state $X_{h,m}$ can transit to the absorbing state if $h = H$, or the maximum level of the skip-list node in each segment $m-1, m-2, \ldots, 1$ is smaller than h. Thus we have:

$$\mathbb{P}\{\otimes \mid X_{h,m}\} = \begin{cases} (1-q^{h-1})^{m-1} & 1 \le h < H, 1 \le m \le M \\ 1 & h = H, 1 \le m \le M \end{cases} \tag{6.4}$$

Equations (6.3) and (6.4) define the transition matrix $\mathbb{T}$ of this absorbing Markov chain. If the states are ordered as $\{X_{1,M}, \ldots, X_{H,M}, X_{1,M-1}, \ldots, X_{H,M-1}, \ldots, X_{1,1}, \ldots, X_{H,1}, \otimes\}$, we can obtain the canonical form of $\mathbb{T}$ as:

$$\mathbb{T} = \begin{bmatrix} \mathbb{T}_1 & \mathbb{T}_2 \\ \underline{\mathbf{0}} & 1 \end{bmatrix}$$

where $\mathbb{T}_1$ is a HM-by-HM matrix describing the probability of transitions between transient states, $\mathbb{T}_2$ is a HM-by-1 column vector describing the probability of transitions from transient states to $\otimes$, and $\underline{\mathbf{0}}$ is a 1-by-HM zero row vector. From equation (6.3), we have:

$$\mathbb{T}_1[(M-m_1)H+h_1][(M-m_2)H+h_2] = \mathbb{P}\{X_{h_2,m_2} \mid X_{h_1,m_1}\} \quad (h_1 < H, h_1 \le h_2, m_1 > m_2)$$

and all other entries in $\mathbb{T}_1$ are zero. The expected number of steps before being absorbed to $\otimes$ when starting from a transient state $X_{h,m}$ is the $((M-m)H+h)$-th entry of the column vector $(\mathbb{I}_{HM} - \mathbb{T}_1)^{-1}\,\underline{\mathbf{1}}$, where $\mathbb{I}_{HM}$ is the HM-by-HM identify matrix and $\underline{\mathbf{1}}$ is a HM-by-1 column vector whose entries are all 1. Obviously, the recovery algorithm starts from a transient state $X_{h,M}$ $(1 \le h \le H)$ with probability $\mathbb{P}\{Y = h\}$. Therefore, the expected cost to recover the skip list header is:

$$\begin{bmatrix} 1-q & q(1-q) & \dots & q^{H-2}(1-q) & q^{H-1} & \underbrace{0 \quad \dots \quad 0}_{(M-1)H} \end{bmatrix} (\mathbb{I}_{HM} - \mathbb{T}_1)^{-1} \, \underline{\mathbf{1}} \times C_{NAND} \tag{6.5}$$

As a result, the expected total overhead to recover from a failure is the summation of the expected costs given in (6.1), (6.2) and (6.5).

6.6 Implementation and Evaluation

In this section, we describe the details of our experiments. HybridStore is implemented in TinyOS 2.1 and simulated in PowerTOSSIMz [82], an accurate power modeling extension to TOSSIM for MicaZ sensor platform. We additionally developed an emulator for a Toshiba TC58DVG02A1FT00 NAND flash (128MB), and a library that intercepts all communications between TinyOS and flash chips (both the NOR and the NAND flash) and calculate the latency and energy consumption based on Table 5.1. With all features included, our implementation requires approximately 16.5KB ROM and 3.2KB RAM, which is well below the limit of most constrained sensor platforms.

The page size and block size of NAND flash are 512B and 16KB, respectively. The segment size of NOR flash varies from 64KB to 256KB. Each Bloom filter section can represent 256 records, with a size of 256B and 3 hash functions. The skip list header buffer is 8B $*$ 10, the size of the address pool is 4B $*$ 15, and the size of a β-Tree bucket is 256B. To improve the performance of insertions as described in Section 6.2.2, the bucket information cache is 10B $*$ 5.

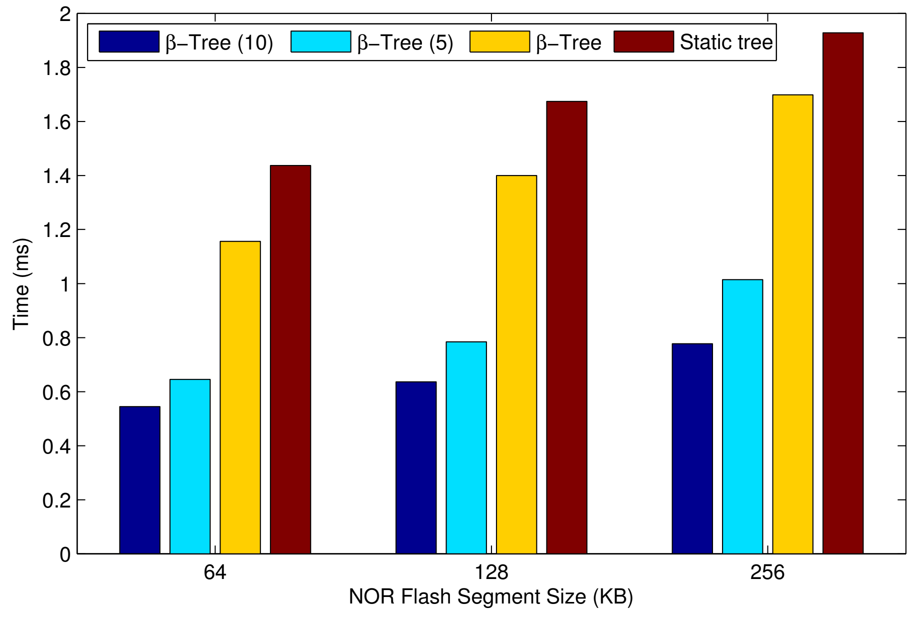
We adopt a trace-driven experimental methodology in which a real dataset is fed into the PowerTOSSIMz simulator. Specifically, we use the Washington Climate Dataset, which is a real dataset of atmospheric information collected by the Department of Atmospheric Sciences at the University of Washington. Our dataset contains $2,630,880$ readings on a per-minute basis between 01/01/2000 and 12/31/2004. Each reading consists of temperature, barometric pressure, etc. We only index the temperature values and use the rest as part of the data records, each of which is of 32 bytes. To simulate data missing (e.g., reading drops due to the long latency during long queries and block erasures, or adaptive sensing), 5% readings are deleted randomly. For each kind of queries, 1000 instances are generated randomly and their average performance is presented here.

We compare HybridStore with MicroHash [9], Antelope [1] and the system in [71]. Since we do not have enough details to reproduce their complete experiments, we directly use the results reported in their papers if necessary. We use the same dataset as MicroHash, and fully implement the static bucket splitting scheme used in [1, 9].

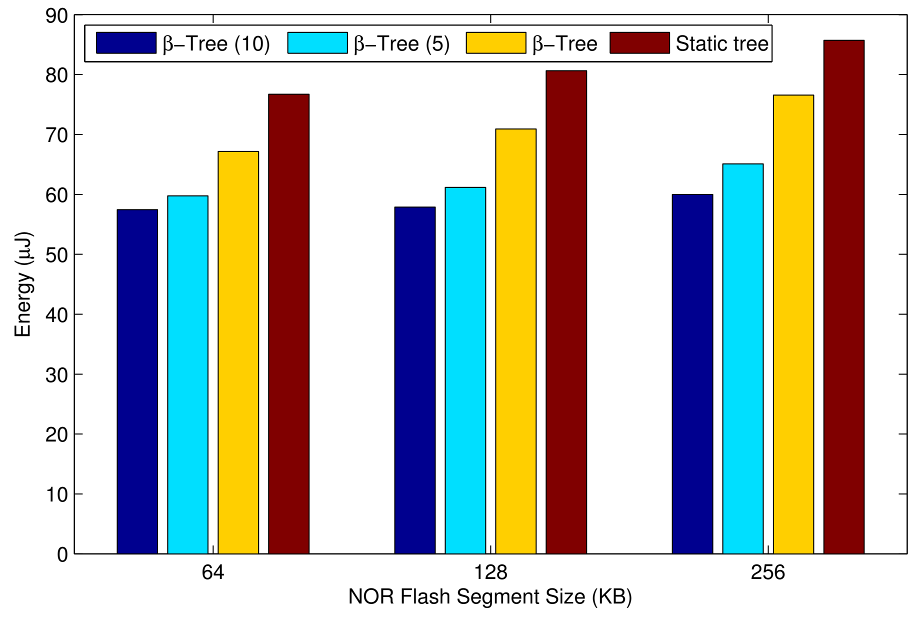
6.6.1 Insertions

We first insert all readings in our dataset into the sensor mote and record the performance of each insertion. Fig. 6.3 shows the average performance of β-Tree and the static bucket splitting scheme used in [1, 9]. Compared to β-Tree, the latter scheme consumes 13.24% more energy, induces 22.72% more space overhead, and results in 18.47% more latency on average if the bucket information cache is disabled. The latency and energy consumption of each insertion approximately equal to the write of 1.31 NAND pages and 0.91 NAND pages, respectively. With a small cache to buffer the information of 5 most recently used buckets, the average energy consumption and latency for each insertion of β-Tree can be further reduced by 13.35% and 42.54%, respectively. We can also observe that the performance can be improved only slightly with a larger cache, because consecutive readings will only be inserted to a few nearby buckets. Therefore, in our final implementation, the bucket information cache is set to be 5 to reduce the overhead.

Fig. 6.4 and Fig. 6.5 show a part of the timeline for insertions at the beginning when the NOR segment size is 64KB with no bucket cache and a cache for 5 bucket information, respectively. Our key observations are as follows. First, although it is very energy-consuming ($26.8mJ$) to transfer the index from NOR flash to NAND flash, it only happens once every 3–4 days and is independent of the data record size. Second, during regular operation, each insertion consumes only $34.4\mu J$. When a reading is not within the current bucket range, the proper bucket can be located or created after traversing about 8–10 bucket headers in β-Tree, even when the current segment is almost full. Since there are about 236 buckets in the β-tree for each segment, our adaptive bucket splitting scheme generates a rather balanced tree. The points corresponding to around $0.15mJ$ (or $0.11mJ$) and $1.2mJ$ additionally include the energy consumption to flush the write buffer to NAND flash and the Bloom filter buffer to NOR flash, respectively. Finally, the bucket information cache



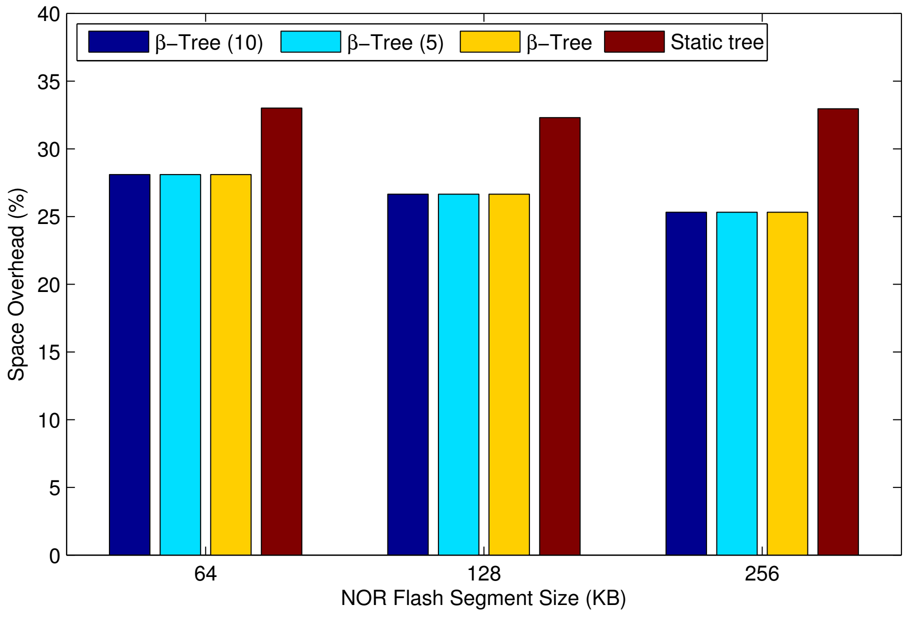
(a) Latency



(b) Energy

Figure 6.3: Performance per insertion

can take advantage of temporal correlation to facilitate the search for the desired β-Tree buckets during insertions efficiently.



(c) Space Overhead

Figure 6.3: Performance per insertion

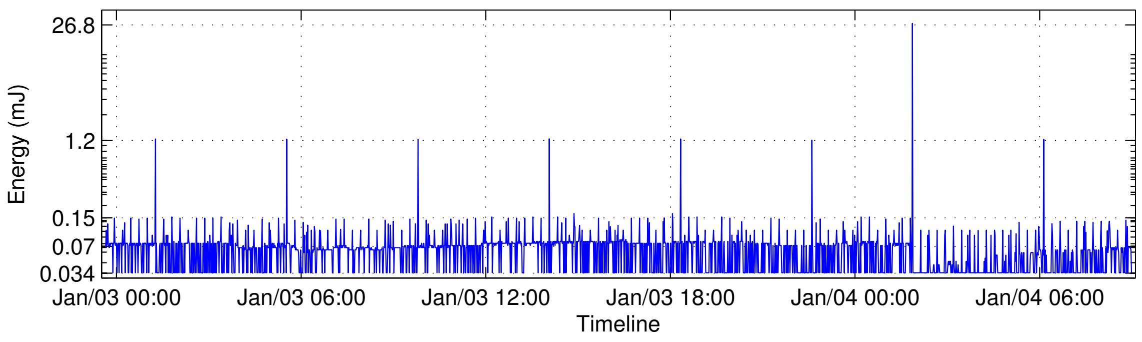


Figure 6.4: Energy consumption of insertions (without bucket information cache)

6.6.2 Time-based Equality Queries

In this experiment, we investigate the performance of time-based equality queries to find a record by its timestamp ($t_1 = t_2$). Fig. 6.6 shows that even though the query time window is quite large (i.e., over 2.5 million readings in 5 years), HybridStore is able to locate the record with about 5–6 page reads. Such a high performance can be achieved due to the following reasons. Firstly, the β-Tree im-

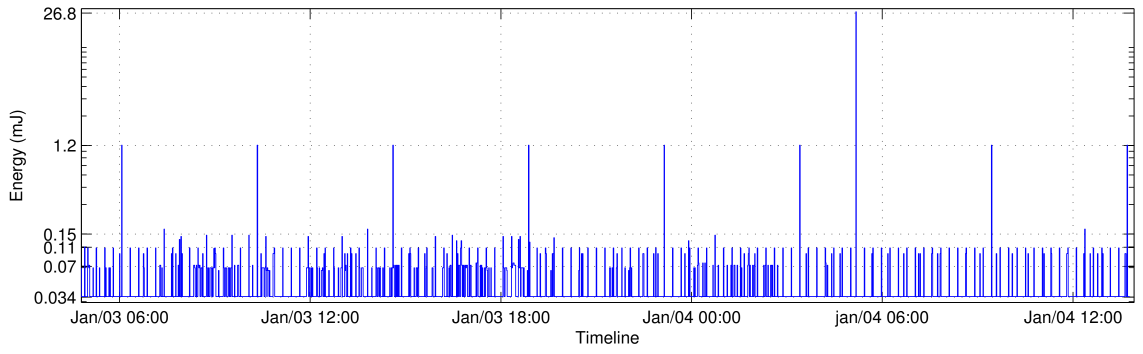
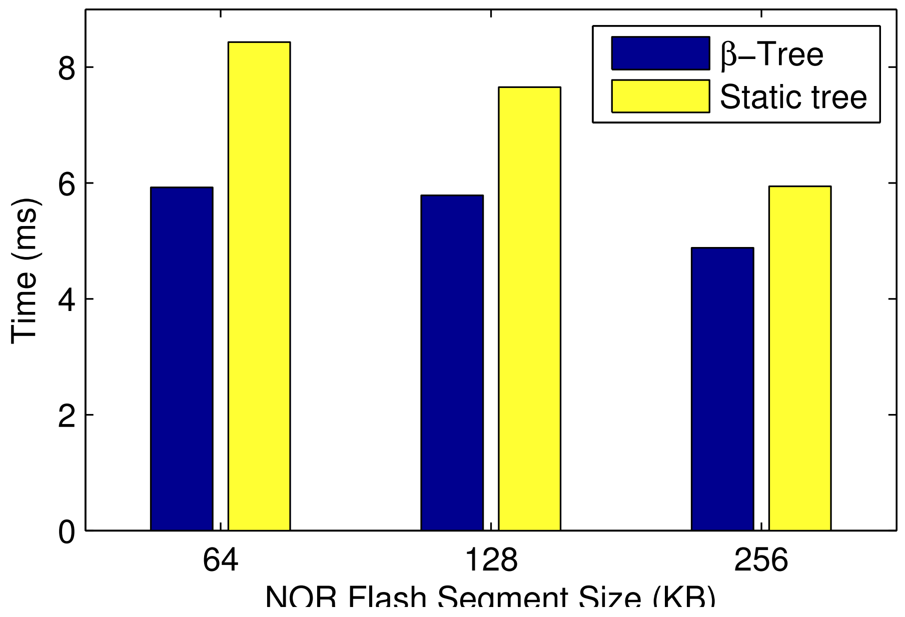


Figure 6.5: Energy consumption of insertions (with a cache for 5 bucket information)

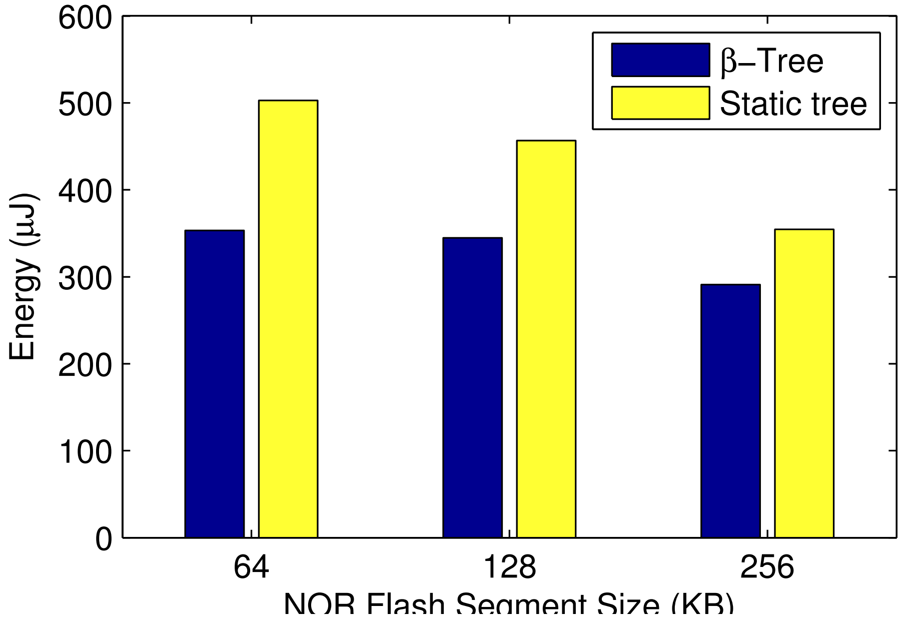
proves the storage efficiency, resulting in fewer segments to store the same number of readings. Secondly, the skip list can locate the segment containing the required timestamp efficiently. Thirdly, the scale binary search can locate the page quickly because all readings are stored continuously in each segment, avoiding traversing through a block chain. Therefore, compared to a global index (e.g., MicroHash requires about 5.4 page reads to process such a query when the buffer size is 2.5KB, as shown in Fig. 14 in [9]), HybridStore has almost the same performance, while consuming less RAM. Compared to Antelope [1], HybridStore can achieve a much better performance if the same dataset is used.

6.6.3 Joint Queries: Time-based Range and Value-based Equality

In this scenario, we study the impact of Bloom filter on joint time-based range and value-based equality queries. Fig. 6.7 shows the average performance per query to search nonexistent key values. We can observe that the in-segment Bloom filter can significantly improve the performance of value-based equality queries (more than 3 times improvement when the NOR segment size is 64KB and the time range



(a) Latency



(b) Energy

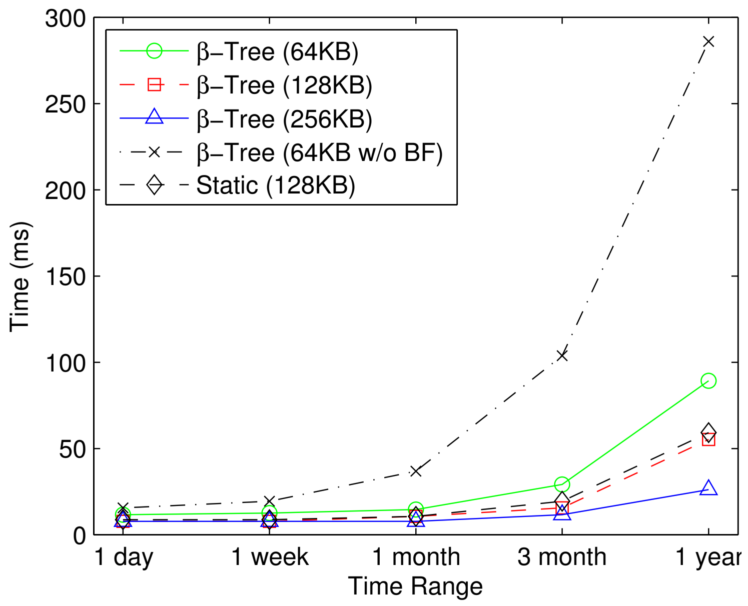
Figure 6.6: Performance of time-equality queries: HybridStore (β-Tree) v.s. Antelope [1]

is more than 3 months). In addition, β-Tree can reduce the latency and energy consumption for queries involving large time window by about $4ms$ and $230\mu J$, respectively. Finally, HybridStore is extremely efficient to check the existence of key values. When the NOR segment size is 256KB, HybridStore can decide the existence of a key value in over 0.5 million readings spanning one year time window in 26.18ms, consuming only 1.56mJ.

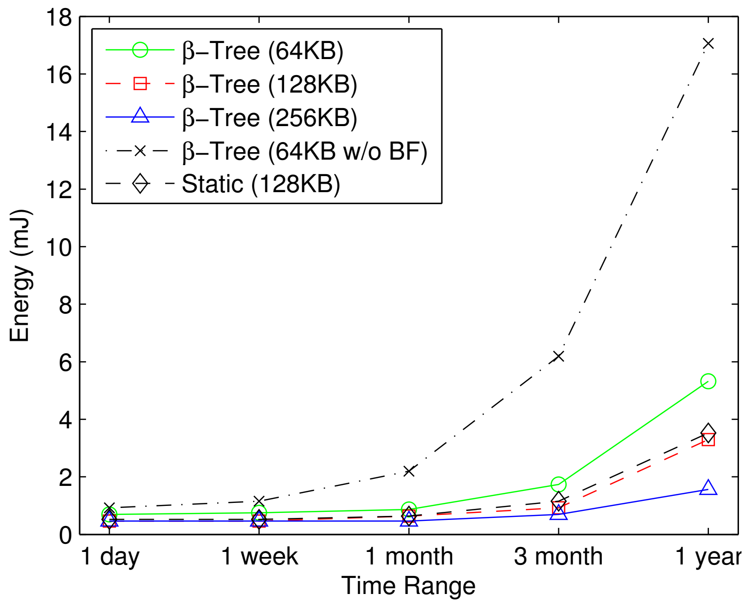
We also investigate the average performance per query to search existing key values, which is shown in Fig. 6.8. The key values vary in $[40\,°\mathrm{F}, 60\,°\mathrm{F}]$. We can observe that the in-segment Bloom filter can reduce the latency and energy consumption for queries involving large time window by 38–116ms and 3–7mJ, respectively. More importantly, HybridStore requires approximately only 826 page reads to get all readings with the given key value in one year time window when the NOR segment size is 256KB. Comparatively, MicroHash requires about 8700 page reads on average to search a given key value $\in [40\,°\mathrm{F}, 60\,°\mathrm{F}]$ in five years time window. Even if we assume that MicroHash can "intelligently" stop searching when a reading below the lower bound of the query time window is encountered, it still requires much more than 1740 page reads for one year time window, because many index pages and data pages are read unnecessarily.

6.6.4 Joint Queries: Both Time-based and Value-based Ranges

In this scenario, we investigate the most common type of queries that involves both time windows and value ranges as selection predicates. Fig. 6.9 shows the average performance per query when the NOR segment size is 64KB. We can observe that HybridStore is extremely efficient to process such queries. When the value range



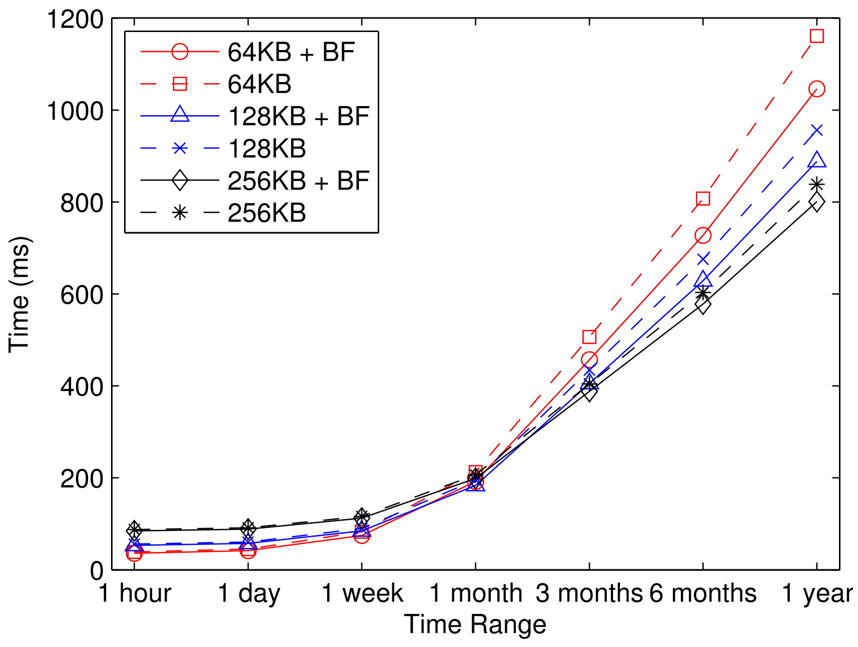
(a) Latency



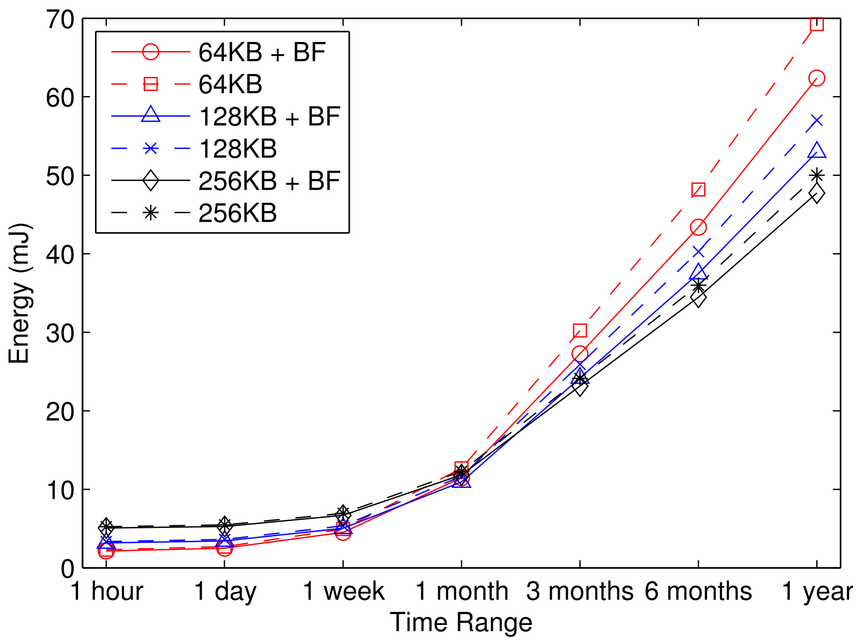
(b) Energy

Figure 6.7: Impact of Bloom Filter on value-based equality queries for nonexistent keys

is 1 °F and the time window is 1 month (typical queries, because readings in small time windows are more interesting), HybridStore can finish the query in 461.6ms,



(a) Latency



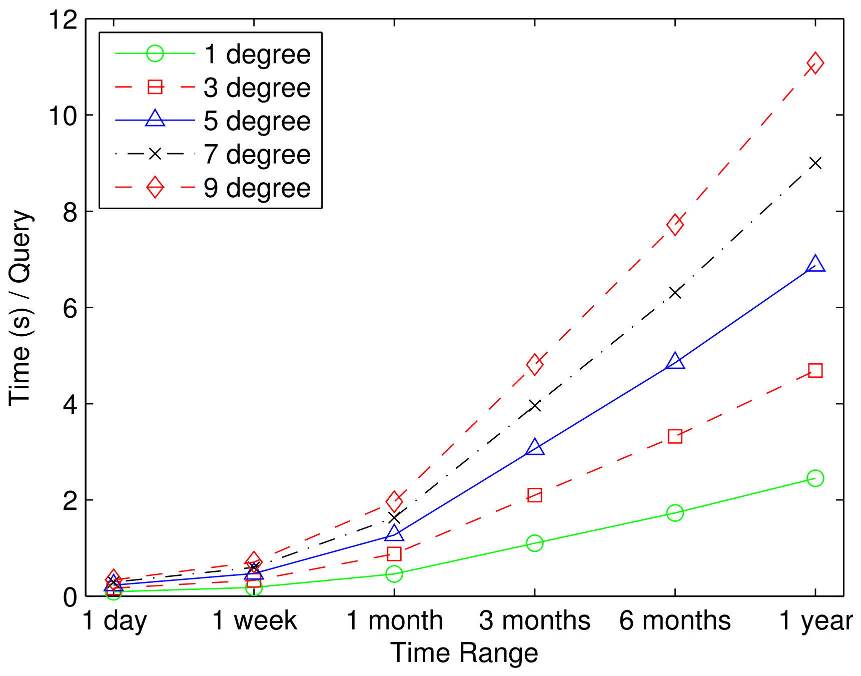
(b) Energy

Figure 6.8: Impact of Bloom Filter on value-based equality queries for existing keys

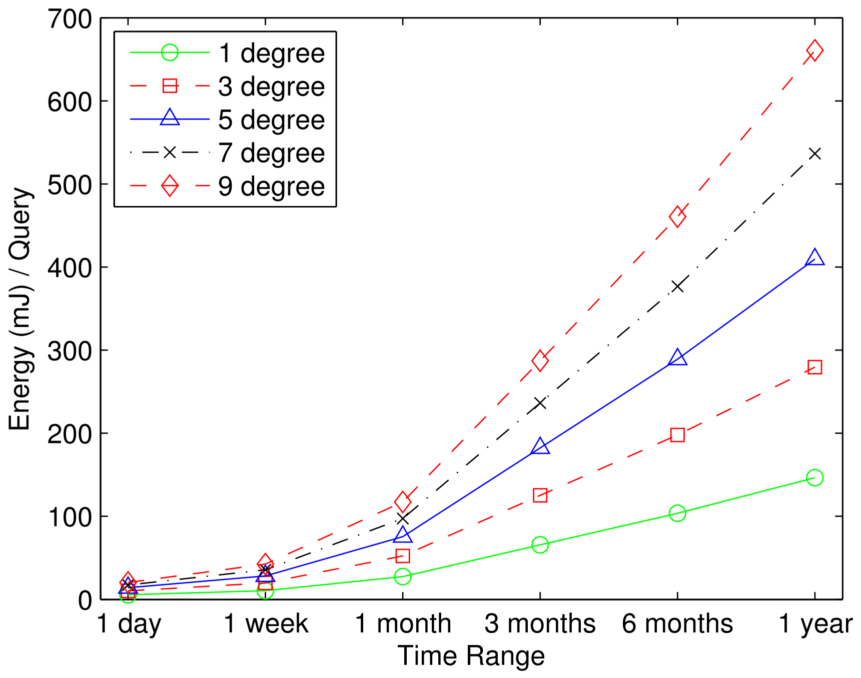
consumes only $27.5mJ$ and returns 2678 readings. For queries involving a large value range (e.g., 9 °F) and a long time window (e.g., 1 year), HybridStore can return $120,363$ readings in $11.08s$, consuming only $660.7mJ$ ($92.04\mu s$ and $5.48\mu J$ per reading on average). Compared to Antelope [1], since the NOR flash is much slower and less energy-efficient, Antelope will take about $20s$ to retrieve 50% readings from a table with only $50,000$ tuples in a range query (shown in Fig. 8 in [1]).

Another index scheme proposed in [71] can support range queries. It will consume about $40mJ$ on average to process a query with 5-degree range on about only $100,000$ readings (about $13,000$ readings are returned). Comparatively, HybridStore will consume $75.61mJ$ to return the same number of readings by processing the same query on more than 2.5 million readings. While our dataset size is 25 times larger than the dataset used in [71], HybridStore consumes only 89% more energy. Therefore, HybridStore is more energy efficient to support queries on large-scale datasets. Besides, the size of each reading in [71] is much smaller, which consists of only a timestamp and a temperature value (12B is enough, while our record size is 32B), resulting in much less data pages. Finally, their scheme requires much more RAM resource (close to 10KB, shown in Page 10 in [71]), because the per-partition B-tree index, interval table, and the *last page of each interval's list* must be maintained in RAM.

The remarkable performance of HybridStore can be explained as follows. Firstly, the inter-segment skip list can skip irrelevant segments outside a query time window and locate the desired segments efficiently, thus avoiding a large number of unnecessary index and data page reads in a global index scheme. Secondly, HybridStore can skip irrelevant segments within the query time window, but storing readings with key values outside the query value range efficiently. Thirdly, our adaptive bucket



(a) Total Latency per query



(b) Total energy per query

Figure 6.9: HybridStore performance per query of full queries

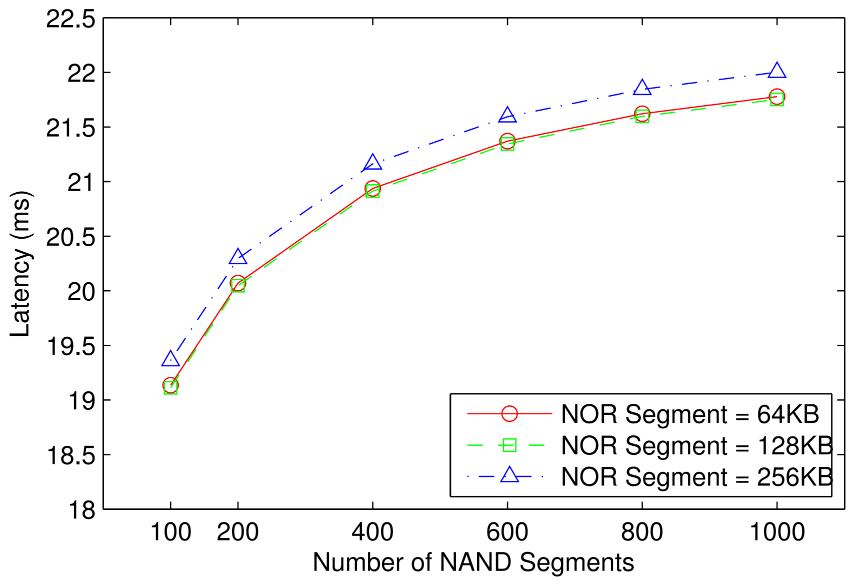
splitting method can create a relatively balanced tree for each segment, thus reducing the number of page reads to retrieve readings from a segment. Furthermore, the in-segment Bloom filter can speed up the existence detection of a key value, while inducing negligible storage overhead (only 1 byte per reading). Finally, the record pool takes advantage of the temporal correlation and spacial locality of sensor readings to further reduce the number of page reads with a small RAM overhead (120B in our case).

6.6.5 Failure Recovery

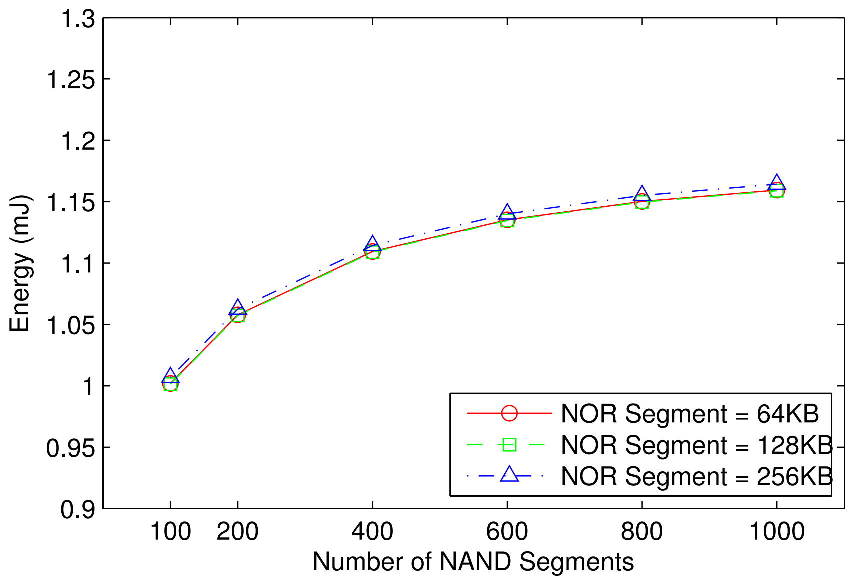
In this section, we calculate the expected overhead for HybridStore to recover from a failure, using the theoretical results from Section 6.5.2. Fig. 6.10 shows that the overhead is very small and increases slightly as the number of stored readings increases significantly. Therefore, our recovery mechanism is very efficient for large-scale data storage.

6.7 Summary

In this chapter, we proposed HybridStore, an efficient data management system for low-end sensor platforms, which exploits both the on-board NOR flash and external NAND flash to store and query sensor data. Compared to existing works that can only support simple queries, HybridStore can process typical joint queries involving both time windows and key value ranges as selection predicates extremely efficiently, even on large-scale datasets. Our evaluation with a large-scale real-world



(a) Latency



(b) Energy

Figure 6.10: Expected overhead of failure recovery

dataset reveals that HybridStore can achieve remarkable performance at a small cost of constructing the index. Therefore, HybridStore provides a powerful new framework to realize *in situ* data storage in WSNs to improve both in-network processing and energy-efficiency.

Chapter 7

HybridDB: An Efficient DB for Incremental ϵ-Approximate Querying

Based on HybridStore, HybridDB provides the support for incremental ϵ-approximate querying that enables clients to retrieve a just-sufficient accuracy level of sensor readings by issuing sub-queries ρ_i sequentially. In this section, we first present an overview of HybridDB with some related basic concepts and algorithms. Then we explain incremental temporal and spatial approximate, and how clients can reconstruct $\widetilde{\Psi}_i$ from $\Delta_1, \Delta_2, \ldots, \Delta_i$. Finally, we describe the adaptive error distribution mechanism to balance trade-offs.

The following interface is provided to issue new queries and update existing queries:

- command uint8_t approxQuery(uint32_t $t_{1,1}$, uint32_t $t_{1,2}$, float k_1, float k_2, float ϵ_1, uint16_t *base*)
- command error_t approxUpdate(uint8_t *queryID*, uint32_t $t_{i,1}$, uint32_t $t_{i,2}$, float ϵ_i)

7.1 Overview of HybridDB

A query Q is started by issuing ρ_1 that will be processed by the `approxQuery` method. HybridDB assigns a unique ID to each query, which is returned to the client as a reference for the following sub-queries. The parameter *base* determines the error distribution between temporal approximate and spatial approximate (Section 7.5). The following sub-queries $\rho_2, \rho_3, \ldots, \rho_\lambda$, either refinement or zoom-in, are issued with the ID of Q and processed by the `approxUpdate` method. Note that the value range $[k_1, k_2]$ is the same for all sub-queries of Q. For simplicity, we assume that for each ρ_i, $t_{i,1}$ is either the timestamp of the *first* reading in some data page or smaller than `System.minT`, while $t_{i,2}$ is either the timestamp of the *last* reading in another data page or larger than the current time. Otherwise, HybridDB will relax the time window $[t_{i,1}, t_{i,2}]$ to satisfy this assumption. In other words, all the readings in every data page that will be loaded by HybridDB during processing ρ_i must be completely covered by $[t_{i,1}, t_{i,2}]$. The reason is that temporal approximate will take all the readings in a data page as a dataset unit (Section 7.2). This assumption is crucial for the correctness of our incremental ϵ-approximate querying algorithm.

HybridDB processes each sub-query with the help of a proxy, which can be a pre-deployed powerful gateway, or a smart phone appearing in the network area *temporarily*. As smart phones have become ubiquitous and powerful mobile computing platforms, the latter choice is preferred in this paper, which can reduce the deployment costs and complexity significantly. Since smart phones can only serve as temporary proxies and different queries may be served by different smart phones, we cannot assume that all the readings that have been retrieved by previous queries are buffered locally on a proxy as in [75]. Instead, a proxy is assumed to have no historical information before processing ρ_1. In addition, we assume that all sub-queries of Q are served by the same proxy. Therefore, clients can issue queries remotely to smart phones, which then return the query results to the clients through

$3G/4G/WiFi$ across the Internet.

Each sub-query ρ_i is processed in two steps. Firstly, the proxy forwards ρ_i' to the sensors, which is formed by modifying the error bound of ρ_i from ϵ_i to $\epsilon_{i,1}$ ($\epsilon_{i,1} \leq \epsilon_i$). Each sensor processes ρ_i' by applying *incremental temporal approximate* locally and transmits the incremental set of readings from HybridStore to the proxy. Incremental temporal approximate is defined as follows:

Definition 7.1 (Incremental Temporal Approximate)**.** Given any subset $\Psi_i[id]$ of Ψ_i ($|\Psi_i[id]| \geq 3$), which is the set of readings contributed by mote id, *temporal approximate* on mote id retrieves a dataset $R_i[id]$ ($R_i[id] \subseteq \Psi_i[id]$), such that $L_\infty(\widehat{\Psi}_i[id], \Psi_i[id]) \leq \epsilon_{i,1}$. Here, $\widehat{\Psi}_i[id] = \{\hat{r}_{id,t} \mid (\forall t \in [t_{i,1}, t_{i,2}]) \wedge (r_{id,t} \in \Psi_i[id])\}$ is constructed using linear approximation as:

$$\hat{r}_{id,t} = \begin{cases} \frac{t-t''}{t'-t''} r_{id,t'} + \frac{t'-t}{t'-t''} r_{id,t''} & r_{id,t} \notin R_i[id] \\ r_{id,t} & o.w. \end{cases} \tag{7.1}$$

in which $r_{id,t'}$ and $r_{id,t''}$ are two readings in $R_i[id]$ with the first and second closest timestamps to $r_{id,t}$. Obviously, $\hat{id} = id$ and $\hat{t} = t$. *Incremental temporal approximate* on mote id returns only $R_i[id] \setminus R_{i-1}[id]$ to the proxy for each ρ_i', and satisfies similar conditions to those in Definition 5.1. The parameter $\epsilon_{i,1}$ is called the *temporal approximate error bound* for ρ_i.

Secondly, after the proxy has received readings from all sensors, it applies *incremental spatial approximate* on them with error bound $\epsilon_{i,2}$, and returns Δ_i to the client. Incremental spatial approximate is defined as follows:

Definition 7.2 (Incremental spatial Approximate)**.** Given any subset $S_i[t] = \{r_{id,t}| (\forall id \in [1,N]) \wedge (r_{id,t} \in \bigcup_{id=1}^{N} R_i[id])\}$ with $|S_i[t]| \geq 3$, which is the set of readings that are generated at time t and returned by all sensors after processing $\rho'_1, \ldots, \rho'_i$, *spatial approximate* on the proxy for time t returns a dataset $\Theta_i[t]$ $(\Theta_i[t] \subseteq S_i[t])$, such that $L_\infty(\widehat{S}_i[t], S_i[t]) \leq \epsilon_{i,2}$. Here, $\widehat{S}_i[t] = \{\hat{r}_{id,t} \mid (\forall id \in [1,N]) \wedge (r_{id,t} \in S_i[t])\}$ is constructed using linear approximation as:

$$\hat{r}_{id,t} = \begin{cases} \frac{id-id_2}{id_1-id_2} r_{id_1,t} + \frac{id_1-id}{id_1-id_2} r_{id_2,t} & r_{id,t} \notin \Theta_i[t] \\ r_{id,t} & o.w. \end{cases} \tag{7.2}$$

in which $r_{id_1,t}$ and $r_{id_2,t}$ are two readings in $\Theta_i[t]$ with the first and second closest IDs to $r_{id,t}$. Obviously, $\hat{id} = id$ and $\hat{t} = t$. *Incremental spatial approximate* for time t returns only $\Theta_i[t] \setminus \Theta_{i-1}[t]$ to the client for each ρ_i, and satisfies similar conditions to those in Definition 5.1. The parameter $\epsilon_{i,2}$ is called the *spatial approximate error bound* for ρ_i.

For both incremental temporal and spatial approximate, a mechanism is needed to compute $R_i[id] \setminus R_{i-1}[id]$ and $\Theta_i[t] \setminus \Theta_{i-1}[t]$ efficiently with little overhead. HybridDB modifies the data shuffling algorithm proposed in [77] to rank readings in $\Psi_i[id]$ or $S_i[t]$ according to their importance, i.e., errors induced if otherwise omitted. In the reordered dataset, readings that will induce larger errors if they are omitted are assigned higher ranks and precedes those inducing smaller errors. The complete procedures to process a general dataset is described in Algorithm 3. We assume that readings in D are sorted in ascending order of t in incremental temporal approximate, and of id in incremental spatial approximate initially. For simplicity, the j^{th} reading in the initial ordered dataset is denoted as r_j, and thus $D = \{r_j\}_{j=1}^{|D|}$. The priority queue always pops out the item with the highest rank, i.e., largest error. The `findFarthest`(D, x, y) function is the same as that in [77], except that the weight of each attribute is included to calculate approximate errors. It returns r_u ($x < u < y$) with the largest error err_u if r_u is approximated using r_x and r_y according to equation (7.1) or (7.2).

```
Algorithm 3 getDelta($D, \epsilon_{old}, \epsilon_{new}$)
Input: Dataset $D = \{r_j\}_{j=1}^{|D|}$, previous error bound $\epsilon_{old}$, and new error bound $\epsilon_{new}$
    ($\epsilon_{old} > \epsilon_{new}$)
Output: The subset of readings that can improve the error bound from $\epsilon_{old}$ to $\epsilon_{new}$
 1: if $|D| \leq 2$ then return ($\epsilon_{old} = \infty$) ? $D : \emptyset$; end if
 2: $result \leftarrow \emptyset$;
 3: if $\epsilon_{old} = \infty$ then $result.add(r_1)$; $result.add(r_{|D|})$; end if          ▷ $|D| \geq 3$
 4: $queue \leftarrow createQueue()$;                        ▷ Initiate the priority queue to be $\emptyset$
 5: $\langle u, err_u \rangle \leftarrow$ findFarthest($D, 1, |D|$);
 6: $queue.enqueue(\langle 1, |D|, u, err_u \rangle)$;
 7: while !$queue.empty()$ do
 8:     $\langle j_1, j_3, j_2, err \rangle \leftarrow queue.dequeue()$;              ▷ Process a new reading
 9:     if $\epsilon_{new} < err \leq \epsilon_{old}$ then $result.add(r_{j_3})$; end if       ▷ $r_{j_3}$ must be returned
10:     if $err \leq \epsilon_{new}$ then break; end if     ▷ Enough readings have been retrieved
11:     if $j_1 + 2 \leq j_2$ then                       ▷ Readings exist between $r_{j_1}$ and $r_{j_2}$
12:         $\langle u, err_u \rangle \leftarrow$ findFarthest($D, j_1, j_2$);
13:         $queue.enqueue(\langle j_1, j_2, u, err_u \rangle)$;
14:     end if
15:     if $j_2 + 2 \leq j_3$ then                       ▷ Readings exist between $r_{j_2}$ and $r_{j_3}$
16:         $\langle u, err_u \rangle \leftarrow$ findFarthest($D, j_2, j_3$);
17:         $queue.enqueue(\langle j_2, j_3, u, err_u \rangle)$;
18:     end if
19: end while
20: return $result$;
```

In each step, our algorithm approximates readings $r_{j_1+1}, r_{j_1+2}, \ldots, r_{j_3-1}$ using r_{j_1} and r_{j_3} to find the reading r_{j_2} with the largest error err (Line 8). If $err \geq \epsilon_{old}$, that means r_{j_2} has already been retrieved in previous sub-queries, and thus it will not be retrieved again. If $err \leq \epsilon_{new}$, that means enough readings have already been retrieved to guarantee the new error bound (Line 10). If $\epsilon_{new} < err \leq \epsilon_{old}$, that means r_{j_2} must be retrieved in order to improve the error bound from ϵ_{old} to ϵ_{new} (Line 9). After adding r_{j_2} to the result, we continue to check if the new error bound can be satisfied, by finding the readings with the largest approximate errors from $r_{j_1+1}, \ldots, r_{j_2-1}$ and $r_{j_2+1}, \ldots, r_{j_3-1}$, respectively (Line 11 to 18). r_1 and $r_{|D|}$ should be retrieved only by the first sub-query, for which $\epsilon_{old} = \infty$ (Line 1 and 3).

From Algorithm 3, we can obtain the following important lemma:

Lemma 7.3. *Given a decreasing sequence $\epsilon'_0, \epsilon'_1, \ldots, \epsilon'_i$ where $\epsilon'_0 = \infty$, and denote the result returned by getDelta$(D, \epsilon'_{j_1}, \epsilon'_{j_2})$ as $D_\Delta[j_1 : j_2]$ $(j_1 < j_2)$, we have $\bigcup_{j=1}^{i} D_\Delta[j-1 : j] = D_\Delta[0 : i]$, $D_\Delta[j_1 - 1 : j_1] \bigcap D_\Delta[j_2 - 1 : j_2] = \emptyset$ for all $j_1 \neq j_2$, and $L_\infty(\widehat{D}, D) \leq \epsilon'_i$, where $\widehat{D}$ is an approximate version of D constructed from $\bigcup_{j=1}^{i} D_\Delta[j-1 : j]$.*

Proof. Firstly, for $\forall r_u \in D_\Delta[0 : i]$, since Algorithm 3 processes it and decides to add it to $D_\Delta[0 : i]$, we can know $\epsilon'_i < err_u < \infty$. Then we must have $\exists j \in [0, i]$ such that $\epsilon'_j < err_u \leq \epsilon'_{j-1}$. That means $r_u \in D_\Delta[j-1 : j]$. Similarly, we can easily prove for $\forall r_u \in \bigcup_{j=1}^{i} D_\Delta[j-1 : j]$, we can obtain $r_u \in D_\Delta[0 : i]$. Therefore, $\bigcup_{j=1}^{i} D_\Delta[j-1 : j] = D_\Delta[0 : i]$.

Assume $D_\Delta[j_1 - 1 : j_1] \bigcap D_\Delta[j_2 - 1 : j_2] \neq \emptyset$ for some j_1 and j_2, i.e., there is at least one reading r_u belongs to both sets. We can learn from Algorithm 3 that $\epsilon'_{j_1} < err_u \leq \epsilon'_{j_1-1}$ and $\epsilon'_{j_2} < err_u \leq \epsilon'_{j_2-1}$, which is impossible because either $\epsilon'_{j_2} < \epsilon'_{j_2-1} \leq \epsilon'_{j_1} < \epsilon'_{j_1-1}$ or $\epsilon'_{j_1} < \epsilon'_{j_1-1} \leq \epsilon'_{j_2} < \epsilon'_{j_2-1}$. Therefore, $D_\Delta[j_1 - 1 :$

$j_1] \bigcap D_\Delta[j_2 - 1 : j_2] = \emptyset$ for all $j_1 \neq j_2$.

Based on the results in [77], $L_\infty(\widehat{D}, D) \leq \epsilon'_i$, where $\widehat{D}$ is an approximate version of D constructed from $D_\Delta[0 : i] = \bigcup_{j=1}^{i} D_\Delta[j-1 : j]$. □

7.2 Incremental Temporal Approximate

The `select` function provided by HybridStore can be modified slightly such that each sensor *id* can process ρ'_i to retrieve all readings in $R_i[id] \backslash R_{i-1}[id]$ efficiently and transmit them to the proxy, which is shown as follows:

- command error_t approxSelect(uint32_t t_1, uint32_t t_2, float k_1, float k_2, float ϵ_{old}, float ϵ_{new})

To reduce the overhead, the Query Processor on each mote is designed to be *stateless*, which has no information about previous sub-queries and thus treats each new sub-query independently. However, the more powerful proxy maintains these information and will provide the right value of ϵ_{old} to sensors, which will be explained in Algorithm 4.

The main difference between `approxSelect` and `select` is how readings are retrieved after a data page is loaded. In `select`, if a reading satisfying the selection predicates is found in the data page, it is transmitted immediately. However, in `approxSelect`, the readings that do not satisfy the selection predicates are first removed from the data page read buffer. The rest readings in the read buffer are ordered in ascending order of t naturally. Then they are passed to Algorithm 3 along with ϵ_{old} and $\epsilon_{i,1}$, and only readings in the result set are transmitted to the proxy. In other words, HybridDB basically takes advantage of the data page read buffer and applies Algorithm 3 to process the readings satisfying the selection predicates in each data page. Although the local NAND flash on each sensor must be accessed again for each sub-query, HybridDB still can reduce the energy consumption extensively with incremental temporal approximate, based on the fact that NAND flash operations are at least two orders of magnitude cheaper than communication.

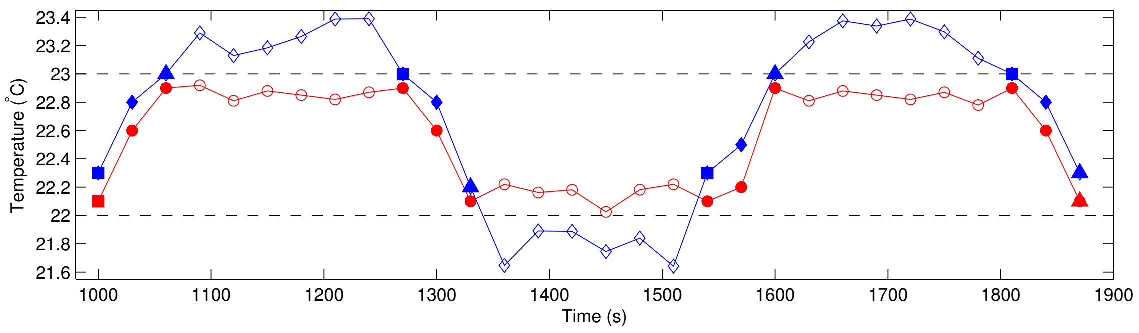


Figure 7.1: Illustration of irregular time windows in incremental temporal approximate: filled points stand for the readings transmitted to the proxy, and square points and triangle points are the first and last readings in the corresponding time window fractions.

HybridDB must solve the problem of irregular time windows brought by value range selection predicates, as illustrated in Fig. 7.1. In the result set, an expected reading $r_{id,t}$ with $t \in [t_1, t_2]$ may be missing for two reasons. Firstly, this reading may be filtered because $key \notin [k_1, k_2]$ (blue trace). Secondly, this reading may be suppressed because $\hat{r}_{id,t}$ recovered from $R_i[id]$ on the proxy using equation (7.1) can guarantee $\|(\hat{r}_{id,t} - r_{id,t}) \times w\|_\infty \leq \epsilon_{new}$ (Line 10 in Algorithm 3, red trace). HybridDB must distinguish these two cases such that a client can reconstruct $\widetilde{\Psi}_i$. Only for the latter case the client needs to recover $\tilde{r}_{id,t}$ and include it in $\widetilde{\Psi}_i$.

This problem is solved by asking sensors to report their time window fractions to the proxy when processing ρ_1 as follows. When each sensor is processing a data page, if all readings in this page satisfy the selection predicates, the first and last readings that will surely be included in the result set are transmitted together in one packet, in which a flag is set. When the proxy receives this packet, it will take the timestamps of these two readings as the beginning and end of a time window fraction with no overhead. Otherwise, this sensor must transmit the information about the set of time window fractions in this page explicitly to the proxy, which consists of the timestamp of the first reading and a bitmap with one bit for each reading in this page to indicate whether it is included in $\Psi_i[id]$. Since each page contains only a few readings generated in a short time, the latter case can occur to only a few pages. With these information, the proxy is able to infer all the time window fractions for each sensor, which are then forwarded to the client along with Δ_1 . Note that for the following sub-query ρ_i $(i = 2, 3, \ldots, \lambda)$, the client can figure out all the time window fractions directly as the intersection of the previous set of time window fractions and $[t_{i,1}, t_{i,2}]$.

7.3 Incremental Spatial Approximate

For each ρ_i, after the proxy has received all the readings in $R_i[id] \setminus R_{i-1}[id]$ from each sensor id, it constructs $S_i[t]$ with readings sorted in ascending order of id for $\forall t \in [t_{i,1}, t_{i,2}]$ from $\bigcup_{id=1}^{N} R_i[id]$, which is then passed to Algorithm 3. Readings in the result set are returned to the client as $\Theta_i[t] \setminus \Theta_{i-1}[t]$. The values of ϵ_{new} and ϵ_{old} to be passed to Algorithm 3 is calculated in Algorithm 4, which describes the implementation of the HybridDB interface on the proxy.

Similarly, a reading $r_{id,t}$ may be missing from $\Theta_i[t] \setminus \Theta_{i-1}[t]$ for two reasons as well: either $r_{id,t} \notin S_i[t]$ because it is not retrieved from mote id, or $r_{id,t} \in S_i[t]$ but it is suppressed by incremental spatial approximate. HybridDB exploits bitmaps to distinguish these two cases. Specifically, for each ρ_i ($i = 1, 2, \ldots, \lambda$), the proxy creates a bitmap B_i with $\left(\frac{t_{i,2}}{\tau} - \frac{t_{i,1}}{\tau} + 1\right) N$ bits if new readings will be retrieved from sensors. If $r_{id,t} \in S_i[t]$, the $\left(\left(\frac{t}{\tau} - \frac{t_{i,1}}{\tau}\right) N + id\right)$-th bit of B_i is set, resulting in a sparse bitmap. To reduce the overhead, the proxy compresses B_i before sending it to the client. The main consideration here is the compression ratio, rather than bitwise operation optimization. Therefore, HybridDB adopts DEFLATE to compress each bitmap, which is a lossless data compression algorithm specified in [83] with high compression ratio. Its implementation is available in the libraries of many programming languages. After the client receives and decompresses B_i, it truncates its local bitmap to retain only the bits corresponding to $[t_{i,1}, t_{i,2}]$, and then carries out the OR bitwise operation on it with B_i to update the local bitmap.

The proxy maintains the status and parameters for each query. For $\rho_2, \rho_3, \ldots, \rho_\lambda$, the proxy needs to validate the parameters (Line 8), and check if this sub-query needs to be processed (Line 9). The proxy processes each ρ_i based on ϵ_i by exploiting the conclusion from Theorem 7.4, which says that the summation of the temporal approximate error $\epsilon_{i,1}$ and the spatial approximate error $\epsilon_{i,2}$ cannot exceed the overall error bound constraint, i.e., $\epsilon_{i,1} + \epsilon_{i,2} \leq \epsilon_i$. If $\epsilon_i < \epsilon_{i-1,1}$, that means more readings

Algorithm 4 Implementation of the HybridDB Interface

```
 1: $Q \leftarrow \emptyset$; $QID \leftarrow 0$;          ▷ $Q$ stores the status and parameters for each query
 2: function APPROXQUERY($t_{1,1}, t_{1,2}, k_1, k_2, \epsilon_1, base$)
 3:   $T_1 \leftarrow t_{1,1}; T_2 \leftarrow t_{1,2}; K_1 \leftarrow k_1; K_2 \leftarrow k_2; \epsilon^* \leftarrow \infty; \epsilon \leftarrow \infty; R \leftarrow \emptyset$;
 4:   $Q[QID] \leftarrow \langle T_1, T_2, K_1, K_2, \epsilon^*, \epsilon, base, R\rangle$;          ▷ Save the query status
 5:   post execute($QID, t_{1,1}, t_{1,2}, k_1, k_2, \epsilon_1$); return $QID$++;     ▷ Begin to process
 6: end function
 7: function APPROXUPDATE($qid, t_{i,1}, t_{i,2}, \epsilon_i$)
 8:   if ($Q[qid]$ == $null$) || ($[t_{i,1}, t_{i,2}] \nsubseteq [Q[qid].T_1, Q[qid].T_2]$) then return $ERROR$; end if
 9:   if $\epsilon_i \geq Q[qid].\epsilon$ then return $SUCCESS$; end if
10:   $Q[qid].T_1 \leftarrow t_{i,1}; Q[qid].T_2 \leftarrow t_{i,2}$;          ▷ Update the query status
11:   post execute($qid, t_{i,1}, t_{i,2}, Q[qid].K_1, Q[qid].K_2, \epsilon_i$); return $SUCCESS$;
12: end function

13: function EXECUTE($qid, t_{i,1}, t_{i,2}, k_1, k_2, \epsilon_i$)
14:   if $\epsilon_i < Q[qid].\epsilon^*$ then          ▷ Must retrieve more data from sensors
15:     $\delta \leftarrow$ splitError($Q[qid], \epsilon_i$);          ▷ Decide $\epsilon_{i,1}$ and $\epsilon_{i,2}$
16:     $B_i \leftarrow newBitmap\left((\frac{t_{i,2}}{\tau} - \frac{t_{i,1}}{\tau} + 1)N\right)$;          ▷ Initiate the bitmap
17:     for $id = 1, 2, \ldots, N$ do     ▷ Retrive $R_i[id] \setminus R_{i-1}[id]$ from each mote $id$
18:       $\langle TWF_{id}, R_\Delta[id]\rangle \leftarrow sensors[id].approxSelect(t_{i,1}, t_{i,2}, k_1, k_2, Q[qid].\epsilon^*, \delta\epsilon_i)$;
19:       if $TWF_{id} \neq \emptyset$ then $radio.send(TWF_{id})$; end if
20:       $Q[qid].R \leftarrow Q[qid].R \bigcup R_\Delta[id]$; $B_i.update(R_\Delta[id])$;
21:     end for
22:     $B_i.compress(); radio.send(B_i)$;
23:   end if
24:   for $t = t_1, t_1 + 1, \ldots, t_2$ do          ▷ Process each $S_i[t]$
25:     $S_i[t] \leftarrow \{r_{id,t} \mid (\forall id \in [1, N]) \wedge (r_{id,t} \in Q[qid].R)\}$;
26:     if $\epsilon_i < Q[qid].\epsilon^*$ then   ▷ Make use of the new data retrieved from sensors
27:       $\Theta_\Delta[t] \leftarrow$ getDelta($S_i[t], \infty, (1-\delta)\epsilon_i$);          ▷ $\epsilon_{i,2} = (1-\delta)\epsilon_i$
28:       $\Theta_\Delta[t].removeDuplicate(); Q[qid].\epsilon^* \leftarrow \delta\epsilon_i$;
29:     else          ▷ Run spatial approximate from last place
30:       $\Theta_\Delta[t] \leftarrow$ getDelta($S_i[t], Q[qid].\epsilon - Q[qid].\epsilon^*, \epsilon_i - Q[qid].\epsilon^*$);
31:     end if
32:     $radio.send(\Theta_\Delta[t]); Q[qid].\epsilon \leftarrow \epsilon_i$;
33:   end for
34: end function
```

must be retrieved from the sensors to improve the error bound (Line 14 – 23). An adaptive error distribution mechanism is adopted to decide $\epsilon_{i,1}$ and $\epsilon_{i,2}$ (Line 15). The `splitError` function returns a value $\delta \in (0, 1]$ based on the parameters of ρ_i, which will be explained in Section 7.5. Note that only the set of readings that can improve the temporal approximate error from $\epsilon_{i-1,1}$ to $\epsilon_{i,1} = \delta\epsilon_i$ is retrieved (Line 18).

For ρ_1, each sensor will report its time window fractions to the proxy as described in Section 7.2. In this case, the proxy needs to forward these information to the client (Line 19). Meanwhile, all retrieved readings will be buffered locally and the bitmap is updated accordingly (Line 20). After all readings in $R_i[id] \setminus R_{i-1}[id]$ have been received from each mote id, the bitmap is compressed and transmitted to the client (Line 22). We can observe that if the proxy can satisfy the error bound of ρ_i using only the readings buffered locally, the overhead to maintain and transmit the bitmap can be eliminated.

Then the proxy applies spatial approximate for every $t \in [t_1, t_2]$ in two different ways based on based on ϵ_i (Line $24 - 33$). If $\epsilon_i < \epsilon_{i-1,1}$, the proxy needs to start over with $(1 - \delta)\epsilon_i$ as the desired error bound (Line 27), because new readings may have been added to $S_i[t]$. The readings that have already been transmitted for previous sub-queries will be removed from the result, and the new $\epsilon_{i,1} = \delta\epsilon_i$ is recorded (Line 28). To facilitate this step, the proxy adds a flag to each reading to indicate whether it has been transmitted to the client. If $\epsilon_i \geq \epsilon_{i-1,1}$, the proxy just needs to carry out spatial approximate to retrieve readings that can improve the error bound from $\epsilon_{i-1} - \epsilon_{i-1,1}$ to $\epsilon_i - \epsilon_{i-1,1}$ (Line 30). Finally, all readings in $\Theta_i[t] \setminus \Theta_{i-1}[t]$ are sent to the client, and the new ϵ_i is recorded (Line 32). Based on Lemma 7.3, we can easily know that $\Delta_i = \bigcup_{t=t_{i,1}}^{t_{i,2}} (\Theta_i[t] \setminus \Theta_{i-1}[t])$, and $\Delta_i \cap \Delta_j = \emptyset$ for all $i \neq j$ $(i, j = 1, 2, \ldots, \lambda)$.

7.4 Construct $\widetilde{\Psi}_i$ on the Client

For each ρ_i ($i = 1, 2, \ldots, \lambda$), the client can construct $\widetilde{\Psi}_i$ in three steps. Firstly, based on Lemma 7.3, the client constructs O_i by retrieving every reading $r_{id,t}$ with $t_{i,1} \leq t \leq t_{i,2}$ from $\Delta_1, \Delta_2, \ldots, \Delta_i$, i.e., $O_i = \{r_{id,t} \mid (\forall r_{id,t} \in \bigcup_{j=1}^{i} \Delta_j) \wedge (t \in [t_{i,1}, t_{i,2}])\}$. Obviously, $\Theta_i[t] = O_i[t]$, where $O_i[t]$ is the subset of readings with the same timestamp t in O_i. Secondly, $\widehat{S}_i[t]$ ($\forall t \in [t_{i,1}, t_{i,2}]$) is constructed from $\Theta_i[t]$ with the help of the bitmap. $\widehat{S}_i[t]$ is initiated with all the readings in $\Theta_i[t]$. Then for any id ($id = 1, 2, \ldots, N$), if $r_{id,t} \notin \Theta_i[t]$ but the $\left(\left(\frac{t}{\tau} - \frac{t_{i,1}}{\tau}\right) N + id\right)$-th bit of its local bitmap is set, HybridDB adds an approximate version $\hat{r}_{id,t}$ to $\widehat{S}_i[t]$, which is constructed according to equation (7.2). Meanwhile, the approximate version $\widehat{R}_i[id]$ of $R_i[id]$ can be constructed as $\widehat{R}_i[id] = \left\{\hat{r}_{id,t} \mid \hat{r}_{id,t} \in \bigcup_{t=t_{i,1}}^{t_{i,2}} \widehat{S}_i[t]\right\}$. Finally, $\widetilde{\Psi}_i[id] = \{\tilde{r}_{id,t} \mid t \textit{ belongs to some time window fraction of mote id}\}$ is constructed as:

$$\tilde{r}_{id,t} = \begin{cases} \frac{t-t''}{t'-t''}\hat{r}_{id,t'} + \frac{t'-t}{t'-t''}\hat{r}_{id,t''} & \hat{r}_{id,t} \notin \widehat{R}_i[id] \\ \\ \hat{r}_{id,t} & o.w. \end{cases} \tag{7.3}$$

which is similar to equation (7.1), except that $R_i[id]$ is replaced with $\widehat{R}_i[id]$. Obviously, $\tilde{id} = id$ and $\tilde{t} = t$. Then, the client can construct $\widetilde{\Psi}_i$ as $\widetilde{\Psi}_i = \bigcup_{id=1}^{N} \widetilde{\Psi}_i[id]$.

Theorem 7.4. *The process described above can construct $\widetilde{\Psi}_i$ correctly, i.e., for any $r_{id,t} \in \Psi_i$, there is a corresponding $\tilde{r}_{id,t} \in \widetilde{\Psi}_i$, and vice verse. In addition, $L_\infty(\widetilde{\Psi}_i, \Psi_i) \leq \epsilon_{i,1} + \epsilon_{i,2}$ is guaranteed.*

Proof. We first prove that $\exists \tilde{r}_{id,t} \in \widetilde{\Psi}_i$ for $\forall r_{id,t} \in \Psi_i$. Obviously, t must belong to a time window fraction of mote id if $r_{id,t} \in \Psi_i$. In addition, from Definition 7.1

and 7.2, we have $\tilde{id} = \hat{id} = id$ and $\tilde{t} = \hat{t} = t$. Thus, $\widehat{R}_i[id] \subseteq \bigcup_{t=t_{i,1}}^{t_{i,2}} \widehat{S}_i[t] \subseteq \widetilde{\Psi}_i$. If $r_{id,t} \in \Theta_i[t]$, $\tilde{r}_{id,t} = \hat{r}_{id,t} = r_{id,t} \in \widehat{R}_i[id] \subseteq \widetilde{\Psi}_i$. Otherwise, $r_{id,t}$ may be suppressed either by spatial approximate or temporal approximate. For the former case, the $\left(\left(\frac{t}{\tau} - \frac{t_{i,1}}{\tau}\right) N + id\right)$-th bit of the bitmap is set, and thus $\hat{r}_{id,t}$ is recovered and added to $\widehat{R}_i[id]$, which means $\tilde{r}_{id,t} = \hat{r}_{id,t} \in \widehat{R}_i[id] \subseteq \widetilde{\Psi}_i$. For the latter case, $\tilde{r}_{id,t}$ will be recovered and added to $\widetilde{\Psi}_i$ based on equation (7.3). Similarly, we can prove that $\exists r_{id,t} \in \Psi_i$ for $\forall \tilde{r}_{id,t} \in \widetilde{\Psi}_i$. Therefore, the approximate version $\widetilde{\Psi}_i$ of Ψ_i is constructed correctly.

Next we prove $L_\infty(\widetilde{\Psi}_i, \Psi_i) \le \epsilon_{i,1} + \epsilon_{i,2}$. For any $r_{id,t} \in \Psi_i$, if the proxy does not apply spatial approximate and returns $S_i[t] \setminus S_{i-1}[t]$ directly, we have $\hat{r}_{id,t} = r_{id,t}$ and $L_\infty(\tilde{r}_{id,t}, r_{id,t}) \le \epsilon_{i,1}$ based on Lemma 7.3. Now due to spatial approximate, $L_\infty(\hat{r}_{id,t}, r_{id,t}) \le \epsilon_{i,2}$ based on Lemma 7.3 as well. In addition, according to Algorithm 3, $R_i[id]$ contains the first and last readings of every data page covered by $[t_{i,1}, t_{i,2}]$. Consequently, in equation (7.3) in the third step, $\tilde{r}_{id,t}$ is approximated using the approximate versions of another two readings (i.e., $\hat{r}_{id,t'}$ and $\hat{r}_{id,t''}$) in the same data page just as that in Algorithm 3, which means $t' < t < t''$. Therefore, for any j, $(j = 3, 4, \ldots, d+2)$, we can obtain:

$$
\begin{aligned}
&|(\tilde{r}_{id,t}[j] - r_{id,t}[j])\, w_j| \le \left| \left(\frac{t - t''}{t' - t''} \hat{r}_{id,t'}[j] + \frac{t' - t}{t' - t''} \hat{r}_{id,t''}[j] - r_{id,t}[j] \right) w_j \right| \\
&\le \left| \left(\frac{t - t''}{t' - t''} \left(r_{id,t'}[j] \pm \frac{\epsilon_{i,2}}{w_j} \right) + \frac{t' - t}{t' - t''} \left(r_{id,t''}[j] \pm \frac{\epsilon_{i,2}}{w_j} \right) - r_{id,t}[j] \right) w_j \right| \\
&\le \left| \left(\frac{t - t''}{t' - t''} r_{id,t'}[j] + \frac{t' - t}{t' - t''} r_{id,t''}[j] - r_{id,t}[j] \right) w_j \right| + \left| \frac{t - t''}{t' - t''} \pm \frac{t' - t}{t' - t''} \right| \epsilon_{i,2} \\
&\le \epsilon_{i,1} + \left| \frac{t - t''}{t' - t''} \pm \frac{t' - t}{t' - t''} \right| \epsilon_{i,2} \le \epsilon_{i,1} + \epsilon_{i,2}
\end{aligned}
$$

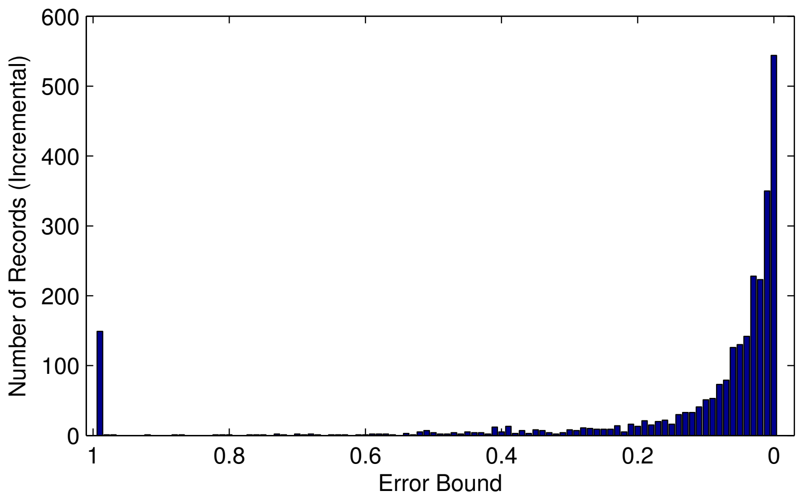
Therefore, $L_\infty(\widetilde{\Psi}_i, \Psi_i) \leq \epsilon_{i,1} + \epsilon_{i,2}$. $\square$

Base on Theorem 7.4, if HybridDB selects $\epsilon_{i,2} = \epsilon_i - \epsilon_{i,1}$ as shown in Algorithm 4, the final overall error bound is guaranteed.

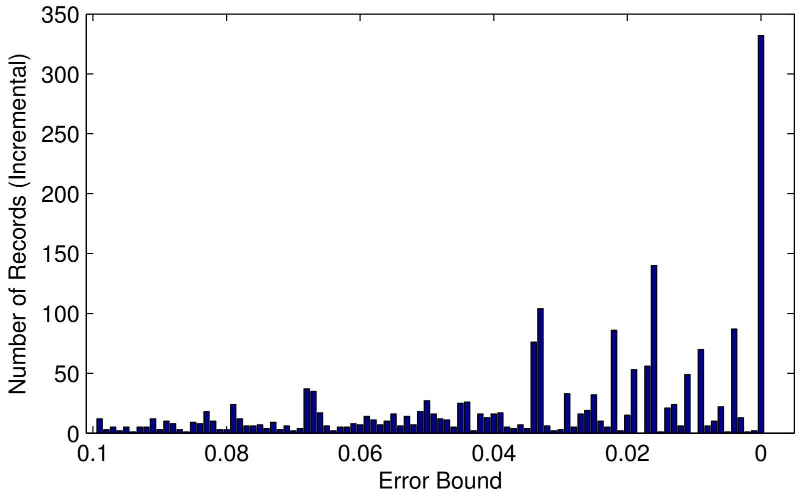
7.5 Adaptive Error Distribution

Algorithm 4 shows that the value of $\epsilon_{i,1}$ decides the trade-offs of energy consumption between sensor motes and the proxy, and response times between ρ_i and $\rho_{i+1}, \rho_{i+2}, \ldots$. Smaller $\epsilon_{i,1}$ means more readings must be retrieved from sensors, and thus larger response time for ρ_i. However, the proxy may be able to satisfy the error bounds of $\rho_{i+1}, \rho_{i+2}, \ldots$ using only the readings buffered locally with higher probability, and thus reduce their response times significantly and eliminate the overhead to compute and transmit the bitmaps. In addition, with larger $\epsilon_{i,2}$, more readings can be suppressed by spatial approximate, which will reduce the number of readings transmitted to the client by the proxy. Therefore, an adaptive mechanism is needed to adjust the error distribution between $\epsilon_{i,1}$ and $\epsilon_{i,2}$ based on the parameters of ρ_i, which is implemented in the `splitError` function by the proxy.

The `splitError` function is designed based on the following observations from the dataset collected from our testbed (Section 7.6). Firstly, temporal correlation between sensor readings is stronger than spatial correlation. Thus, with the same error bound, temporal approximate can potentially suppress more readings than spatial approximate. Secondly, the cardinality of $R_i[id] \setminus R_{i-1}[id]$ is very small with large $\epsilon_{i,1}$, but increases dramatically as $\epsilon_{i,1}$ becomes small, as shown in



(a) Error step: 0.01



(b) Error step: 0.001

Figure 7.2: Distribution histograms of the readings from Sensor 7

Fig. 7.2a. In addition, with small $\epsilon_{i-1,1}$, the cardinality of $R_i[id] \setminus R_{i-1}[id]$ can still be large even with a very tiny improvement on the error bound (i.e., $\epsilon_{i-1,1} - \epsilon_{i,1}$ is very tiny), as shown in Fig. 7.2b. Therefore, `splitError` should return a larger value to favor temporal approximate for large $k_2 - k_1$, large $t_{i,2} - t_{i,1}$ or small ϵ_i. In our implementation, the following expression is adopted:

$$\delta = \left(\max\left(\frac{base}{t_{i,2} - t_{i,1}} \times \frac{c_1}{k_2 - k_1}, 1\right)\right)^{-\min(c_2\epsilon_i,\ c_3)} \in (0, 1] \tag{7.4}$$

where *base* is an adjustable parameter inputted by the client, and c_1, c_2 and c_3 are application-specific positive constants.

7.6 Implementation and Evaluation

In this section, we describe the details of our experimental methodology. We have implemented HybridDB in TinyOS 2.1 and evaluated incremental ϵ-approximate querying in the network level. With all features included, our implementation requires approximately 22.5KB ROM and 3.76KB RAM, which is well below the limit of most constrained sensor platforms. All parameters of the HybridStore implementation are the same as those in Section 6.6.

We investigate the performance of HybridDB from a networking perspective, evaluating the efficiency of incremental ϵ-approximate querying for both typical refinement queries and zoom-in queries in a real testbed. Our experiment testbed consists of 12 IRIS sensor motes, one iMote2 mote as the proxy, and another iMote2 mote as the client that is connected to a PC through a serial port to receive queries and report experiment statistics. The testbed is deployed in the north wing of the Engineering Annex building in our campus, as shown in Fig. 7.3, to sample temperature, relative humidity and light once every 30 seconds. Each IRIS mote partitions its flash memory into two volumes: the first volume of 64KB is used as the NOR flash, and the second volume of 448KB emulates a Toshiba TC58DVG02A1FT00 NAND flash. Each IRIS mote transmits two readings in every packet, while the proxy puts four readings together into each packet. In the case of multi-hop communication, each IRIS mote only records the number of packets sent by itself, and does not count the packets forwarded for other motes. To ensure reliable data collection, a packet will be retransmitted upon failure.

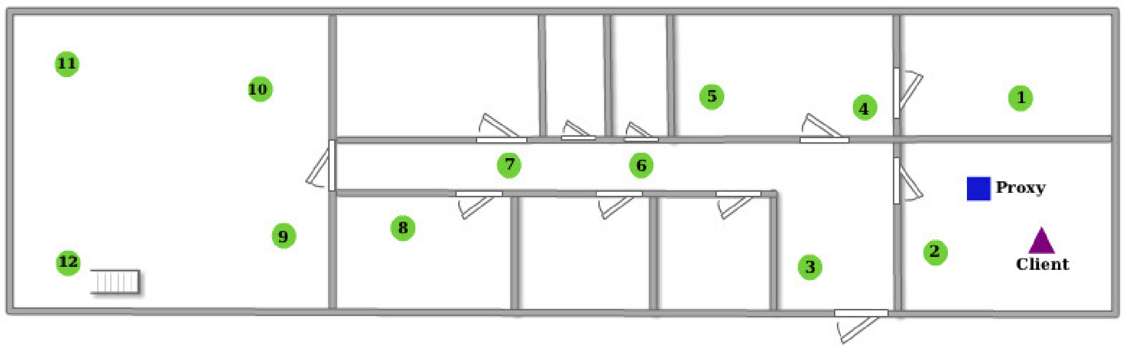
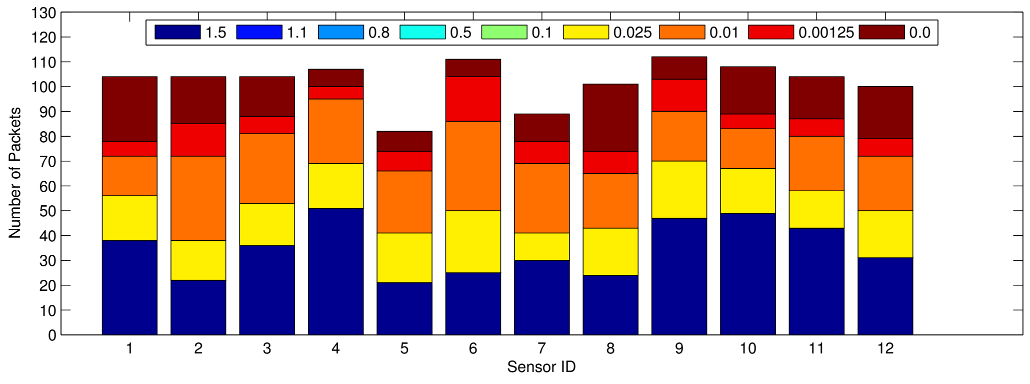


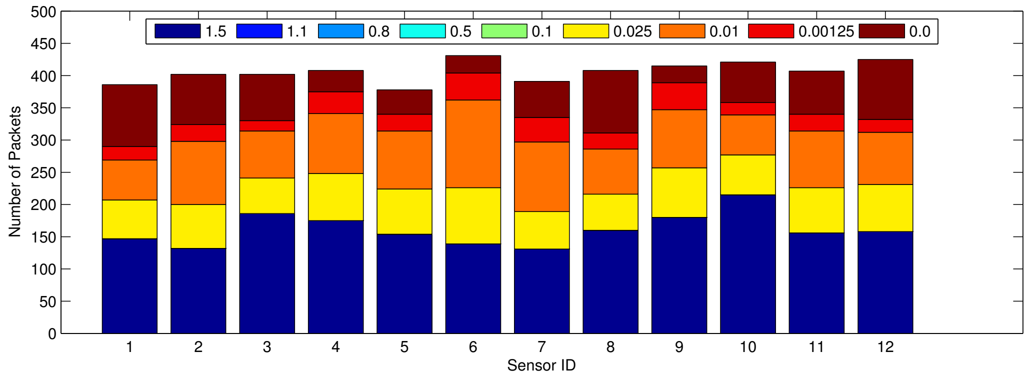
Figure 7.3: Testbed deployment in the Engineering Annex building: both single-hop and multi-hop networks are considered. In the latter case, mote 1 – 8 locates within one hop of the proxy, while mote 9 – 12 communicate with the proxy through two-hop paths.

We first investigate the performance of HybridDB to process refinement queries. Two queries are issued to the proxy by the client, with a short time window (2 hours) and a long time window (8 hours), respectively. The value range for both queries is $[20\,°\mathrm{C}, 25\,°\mathrm{C}]$, and the error bounds of their sub-queries decrease from 1.5 to 0. Other parameters are configured such that $\delta = 10^{-\epsilon_i}$ after substituting them into equation (7.4). The statistics of the number of packets sent by each IRIS mote and the proxy are shown Fig. 7.4 and Fig. 7.5, respectively.

We can observe that the test results of both refinement queries show similar characteristics. Firstly, both the sensors and the proxy need to transmit many packets for ρ_1, because they need to transmit not only sensor readings, but also



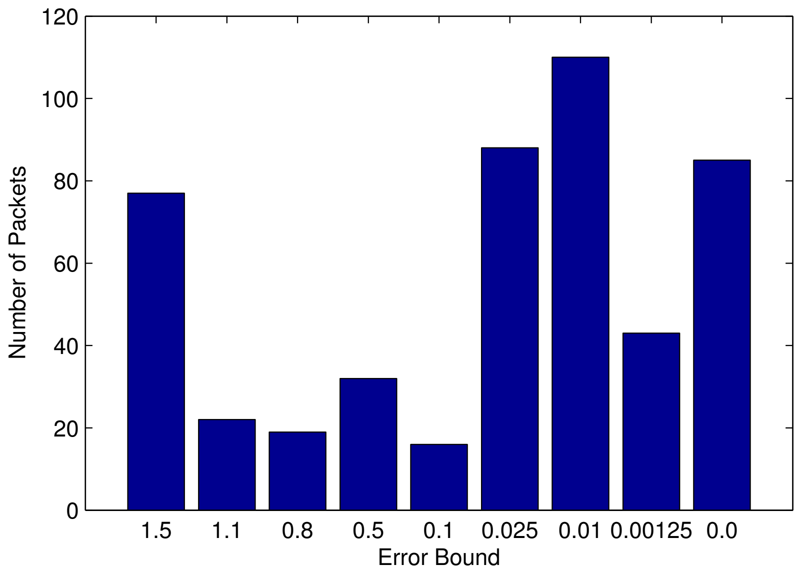
(a) Time window = 2 hours



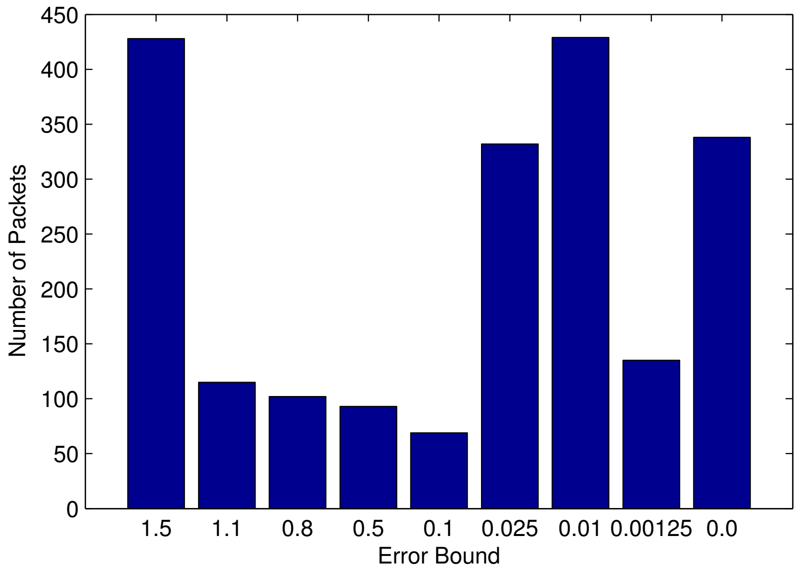
(b) Time window = 8 hours

Figure 7.4: Number of packets sent by each sensor in each sub-query

the information about time window fractions on each sensor and the bitmap. Due to the relatively large number of readings, the compression ratio of the bitmap on the proxy is not so good, resulting in more overhead. However, since ϵ_1 is large, our adaptive error distribution mechanism predicts that the client is very likely to issue more refinement sub-queries in order to get a satisfactory set of readings. As a result, a small δ is returned such that more readings can be retrieved from the sensors in advance. We can see that the proxy processes the following four sub-queries directly using only the readings buffered locally. This can not only reduce the response times for these sub-queries significantly as shown in Fig. 7.6, but also



(a) Time window = 2 hours



(b) Time window = 8 hours

Figure 7.5: Number of packets sent by the proxy in each sub-query

eliminate the overhead to transmit bitmaps to the client because they do not require the proxy to retrieve more readings from the sensors. Finally, as ϵ_i becomes very small, a large δ is returned to force the proxy to retrieve only the required readings from the sensors, because the client may be satisfied with the output of ρ_i with high probability. Meanwhile, as shown in Fig 7.2b, only a minor improvement over a tiny error bound may result in a large amount of extra readings to be retrieved.

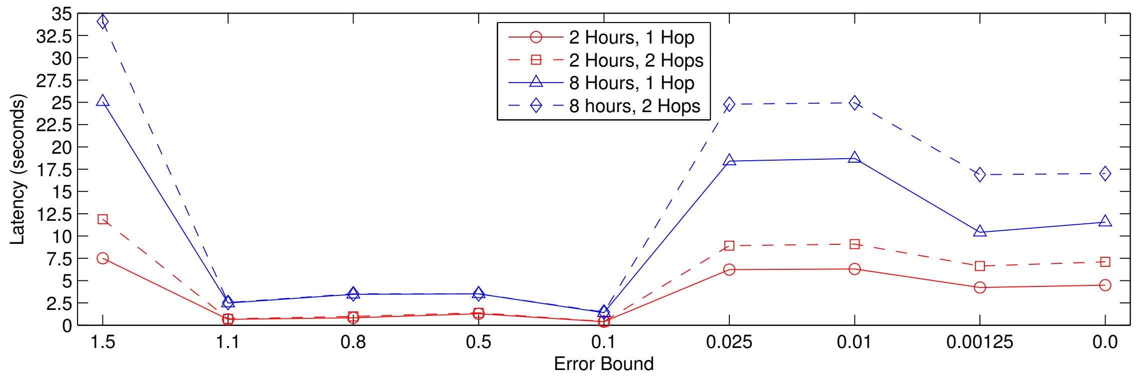


Figure 7.6: The response time of each sub-query with different query time windows in different network topologies

With the help of our adaptive error distribution mechanism, the response times for sub-queries with small error bounds are well balanced. As a result, HybridDB can also provide a much better user experience, because the client can refine the query result gradually with an acceptable response time for each improvement, rather than wait for a long time at the risk of no response from the proxy, or retrieving over-qualified set of readings. Fig. 7.6 also shows that HybridDB works well in the multi-hop network, with the average response time increased by 41.61% compared to that in the single-hop network.

Next, we investigate the performance of HybridDB to process zoom-in queries, which are especially useful for searching interesting events in large time windows and retrieving only a small amount of readings to get detailed information about these events. In this scenario, the parameters in equation (7.4) are configured such that $\delta = \left(\frac{12\ hours}{t_{i,2}-t_{i,1}}\right)^{-\epsilon_i}$. We imitate a typical procedure that may be adopted by the client to locate interesting events. The client first issue three sub-queries with large time windows and large error bounds to obtain an overview of the environment to learn the general situation. Suppose the client finds a time interval in which some interesting events potentially happened. Then, the client issues the following zoom-in sub-queries with tighter error bounds and smaller time windows to focus on the interesting events and obtain more detailed information gradually. The test result is shown in Fig. 7.7, which reveals that HybridDB can process each zoom-in sub-queries promptly. The client can locate the interesting events efficiently with highly or even completely accurate information, while significantly reducing the amount of energy wasted to retrieve unnecessary details in blind event searching and avoiding depressing large latency.

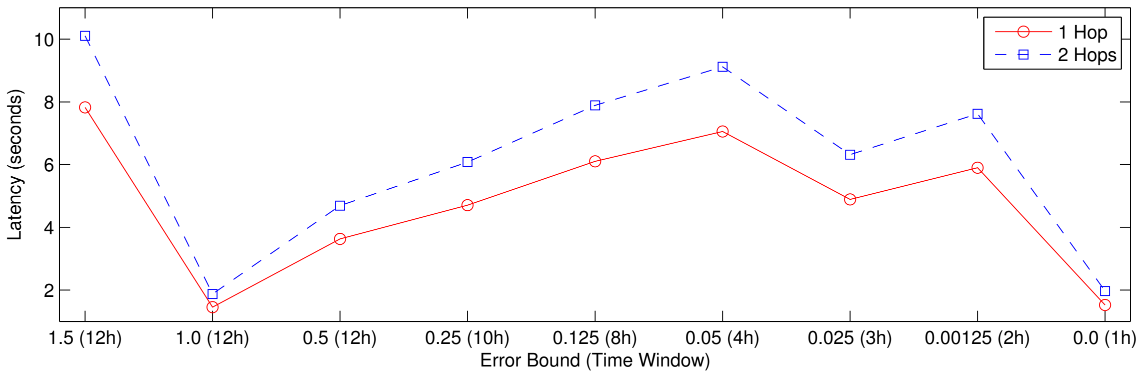


Figure 7.7: The response time of a sequence of Refinement-ZoomIn sub-queries in different network topologies

7.7 Summary

In this chapter, we proposed HybridDB, an efficient light-weight distributed database system for flashed-based storage-centric wireless sensor networks. HybridDB supports an incremental ϵ-approximate querying scheme, with which clients can carry out "Google Map"-like refinement queries and zoom-in queries to locate interesting events and retrieve just-sufficient set of readings with arbitrary L_∞-norm error bounds gradually. Unlike traditional approximate querying mechanisms, sensor readings that have been already retrieved by previous sub-queries with looser error bounds will not be transmitted again. An adaptive error distribution mechanism between temporal approximate and spatial approximate is designed based on the characteristics of sensor readings and typical query procedures. This mechanism can balance the trade-offs of energy consumption between sensors and the proxy, and response times between the current sub-query and the following sub-queries effectively.

Our implementation and evaluation of HybridDB on our sensor testbed deployed in real world reveals that HybridDB can process both refinement queries and zoom-in queries efficiently, providing much better user experience by processing each sub-query promptly, and significantly reducing the amount of energy wasted to retrieve unnecessary readings that will generate over-qualified results.

Chapter 8

Conclusions

In this dissertation, we designed and implemented a model-based systems design framework, called WSNDesign, to facilitate the design and implementation of wireless sensor networks for Smart Buildings. WSNDesign provides a hierarchy of model libraries to model various behaviors and structures of sensor networks in the context of Smart Buildings, and introduces a system design flow to compose both continuous-time and event-triggered modules to develop applications. WSNDesign can support co-simulations of SysML and Simulink with the help of IBM Rhapsody. In addition, based on the FMI standard, WSNDesign can generate codes and FMUs to co-simulate TinyOS applications and Modelica models. Finally, WSNDesign can expose a sequence of design choices to system designers, and provides instant feedback about the influence of a design decision on the complexity of system analysis.

To enrich our model libraries and facilitate the collection and retrieval of sensor data for application development, we designed and implemented HybridDB, a distributed database system supporting *in situ* data storage on sensor motes and ϵ-approximate querying in sensor networks. We implemented HybridDB in TinyOS 2.1 and studied its performance on a sensor network testbed. The implementation of HybridDB is transformed and imported to WSNDesign as a part of the Service Library.

Appendix A

Formulation of Tree Decomposition

Definition A.1. Define a *system* as the tuple $\mathcal{P} = \langle \mathcal{L}, \mathcal{P}_1(\mathcal{X}_1), \ldots, \mathcal{P}_{\mathcal{M}}(\mathcal{X}_{\mathcal{M}})\rangle$, with $\mathcal{L} = \{\Sigma_1, \ldots, \Sigma_{\mathcal{N}}\}$ and $\mathcal{X}_i \subseteq \mathcal{L}$ for $i = 1, \ldots, \mathcal{M}$. Each $\Sigma_i \in \mathcal{L}$ is a set corresponding to the domain of a system variable x_i. Each $\mathcal{P}_i$ $(i = 1, \ldots, \mathcal{M})$ is a general component that influences the variables with domains $\mathcal{X}_i$.

Observe that in general, the $\mathcal{X}$ values are not disjoint. In this model, the sharing of variables between components indicates communication between those components. We may exploit a Parametric diagram in SysML to capture any system as defined above. The constraint blocks are the components $\mathcal{P}_i$ of the system and the variables correspond to the variables Σ_i of the system. Each constraint block $\mathcal{P}_i$ has its own associated list of variables $\mathcal{X}_i$.

Definition A.2. Define the *flattening* of a system $\mathcal{P}$ as the graph $G = \langle \mathcal{L}, E\rangle$ with $E = \{(x, y) \mid \exists i \in [1, \mathcal{M}]\ s.t.\ (x, y \in \mathcal{X}_i) \wedge (x \neq y)\}$. Every parameter set defined by $\mathcal{X}_i$ induces a clique of mutually connected nodes in the flattened graph G.

Definition A.3. Define the *elimination* of a node $\Sigma \in \mathcal{L}$ from the graph $G = \langle \mathcal{L}, E\rangle$, denoted by $\bigoplus_{\Sigma} G$, as the graph $G' = \langle \mathcal{L} \setminus \{\Sigma\}, E'\rangle$, where E' is defined as $(E \setminus \{(x, y) \mid \Sigma \in \{x, y\}\}) \bigcup F$. Here, F is the set of links in the clique induced by the set of neighbors $N(\Sigma)$ of Σ.

This definition reflects the fact that in a semiring context, eliminating a variable (by summation) first entails collecting all the constraints that include this variable. The induced clique over the neighbors of Σ affects the necessary collection of constraints. Let $\mathcal{L}_p = \langle \Sigma_{i_1}, \ldots, \Sigma_{i_\mathcal{M}} \rangle$ be a permutation of $\mathcal{L}$, which can be viewed as an elimination ordering.

Definition A.4. The *sequence of graphs induced by an elimination ordering* $\mathcal{L}_p$, which is denoted by $\langle G_1(\mathcal{L}_p), \ldots, G_{\mathcal{N}+1}(\mathcal{L}_p) \rangle$, is defined as $G_1(\mathcal{L}_p) = G$ and $G_{k+1}(\mathcal{L}_p) = \bigoplus_{\Sigma_{i_k}} G_k(\mathcal{L}_p)$ for $k = 1, \ldots, \mathcal{N}$.

It is clear from the above definition that $G_{\mathcal{N}+1}(\mathcal{L}_p)$ must be an empty graph because all nodes have been eliminated. This elimination induces also a sequence of cliques in the graph.

Definition A.5. The *sequence of cliques induced by an elimination ordering* $\mathcal{L}_p$, which is denoted by $\langle C_1(\mathcal{L}_p), \ldots, C_\mathcal{N}(\mathcal{L}_p) \rangle$, is defined as $C_k = N_{G_k(\mathcal{L}_p)}(\Sigma_{i_k}) \bigcup \{\Sigma_{i_k}\}$ for $k = 1, \ldots, \mathcal{N}$. Here, $N_{G_k(\mathcal{L}_p)}(\Sigma_{i_k})$ is the set of neighbors of Σ_{i_k} in $G_k(\mathcal{L}_p)$ from the sequence of graphs induced by $\mathcal{L}_p$.

Note that in this definition, there may be cliques $C_k(\mathcal{L}_p)$ that are contained in other cliques.

Definition A.6. The *width* of graph G with respect to ordering $\mathcal{L}_p$ is defined as the maximum size of the cliques in the sequence of induced cliques minus 1, i.e., $W_G(\mathcal{L}_p) = \max_k |C_k(\mathcal{L}_p)| - 1$.

The extra minus 1 ensures that the treewidth of a tree is 1. The *treewidth* W

of a system $\mathcal{P}$ is defined as:

$$W = \min_{\mathcal{L}_p} W_G(\mathcal{L}_p) = \min_{\mathcal{L}_p} \max_k |C_k(\mathcal{L}_p)| - 1 \tag{A.1}$$

The value of W gives the minimal tree decomposition of the system. The optimization in equation (A.1) is NP-hard [84]. Since such problems are not tractable in general, we present a hierarchical algorithm exploiting random search to compute a sequence of upper bound on treewidth.

A node is *simplical* if all of its neighbors are mutually connected. Since the optimization parameter is a permutation, the search space has a tree structure. There are sub-sequences of the permutation that are determined and do not need to be searched. Specially, when there are simplical nodes in the network, they can be eliminated immediately. Furthermore, when there are multiple simplical nodes, the order of their eliminations does not affect the resulting treewidth. A simplical node can be eliminated by removing it from the set of nodes and all the links incident to it. Given two simplical nodes x and y, the resulting graph after eliminating both of them from $G = \langle \mathcal{L}, E \rangle$ has the nodes $\mathcal{L} \setminus \{x, y\}$ and the set of edges that are not incident to x or y. Therefore, the eliminating order of x and y does not matter to the resulting graph, i.e., $\bigoplus_x \bigoplus_y G = \bigoplus_y \bigoplus_x G$.

Theorem A.7. *Let $\mathcal{L}_p$ be an elimination order where Σ_{i_k} and Σ_{i_k+1} are both simplical in $G_k(\mathcal{L}_p)$, and $\mathcal{L}'_p$ be another elimination order by swapping Σ_{i_k} and Σ_{i_k+1} in $\mathcal{L}_p$. Then $W_G(\mathcal{L}'_p) = W_G(\mathcal{L}_p)$.*

Proof. Since $\mathcal{L}_p$ and $\mathcal{L}'_p$ differ only in swapping Σ_{i_k} and Σ_{i_k+1}, we have $G_i(\mathcal{L}_p) = G_i(\mathcal{L}'_p)$ for $i = 1, \ldots, k$. In addition, since both Σ_{i_k} and Σ_{i_k+1} are simplical, we can obtain $G_i(\mathcal{L}_p) = G_i(\mathcal{L}'_p)$ for $i = k+3, \ldots \mathcal{N}+1$ from the above discussion. Now we consider the cases for $i = k+1, k+2$. If Σ_{i_k} and Σ_{i_k+1} are neighbors, the fact that they are simplical in $G_k(\mathcal{L}_p)$ implies that Σ_{i_k} and Σ_{i_k+1} are part of the same clique in $G_k(\mathcal{L}_p)$. So Σ_{i_k} and Σ_{i_k+1} are symmetrical and indistinguishable with respect to the sizes of the cliques formed. If they are not neighbors, it is clear that $C_{i+1}(\mathcal{L}_p) = C_{i+2}(\mathcal{L}'_p)$ and $C_{i+2}(\mathcal{L}_p) = C_{i+1}(\mathcal{L}'_p)$ because the eliminations are completely independent of each other. Therefore, we have $W_G(\mathcal{L}'_p) = W_G(\mathcal{L}_p)$. $\square$

Theorem A.7 states that eliminations of simplical nodes are commutative with respect to the treewidth of the resulting graphs. Therefore, simplical nodes can be eliminated in any order without impacting the width of the graph.

Interactive Tool Based on the theory of tree decomposition, we develop a graphical tool that exposes a sequence of design choices to system designers, provides instant feedback about the influence of a design decision on the complexity of system analysis, and gradually reduce the upper bound on system treewidth over time. The algorithm to find the treewidth of a graph can be sketched out as follows:

1. Eliminate all simplical nodes in any order.

2. If any nodes remain, eliminate one randomly.

3. If any nodes remain, return to step 1.

This algorithm eventually finds all valid elimination orders if running enough iterations. The probability that a particular order is not found is roughly $\left(1 - \frac{1}{n}\right)^k$, where n is the number of all valid elimination orders and k is the number of iterations.

In order to speed up convergence of the above algorithm, we collect statistics about the decisions made and use these statistics to improve future guesses. The algorithm that we implement builds a tree of pre-specified *bounded* size, as shown in Fig. A.1. The system uses a predefined size for the search tree, bounded by memory constraints. At each node of the tree, a metric representing the sample mean under blind search from that node is maintained. The algorithm alternates between blind search and directed search. The blind search samples alternatives uniformly. The directed search samples branches in proportion to the expected value of the score function, which in this case is e^{-w} where w is the expected width of a branch.

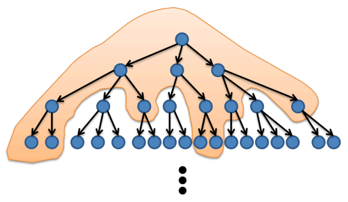


Figure A.1: This figure shows the search tree through the permutation space. The nodes are the different valid eliminations. The shaded region represents the nodes held in memory for which statistics have been collected.

Appendix B

Publication

[1] **B. Wang** and J. S. Baras. "WSNDesign: A Modeling, Design and Co-simulation Framework for Wireless Sensor Networks". *In preparation for journal submission*, 2013.

[2] Book chapter in "System Design, Modeling, and Simulation Using Ptolemy II" (Editor: Claudius Ptolemaeus), *Ptolemy.org*, 2014

[3] **B. Wang** and J. S. Baras. "HybridSim: A Modeling and Co-simulation Toolchain for Cyber-physical Systems ". *In the 17th IEEE/ACM International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, 2013, Delft, Netherlands.

[4] **B. Wang**. "Storage-centric Sensor Networks for Smart Buildings". *In the 12th ACM/IEEE Conference on Information Processing in Sensor Networks (IPSN, extended abstract)*, 2013.

[5] **B. Wang** and J. S. Baras. "HybridDB: An Efficient Database System Supporting Incremental ϵ-Approximate Querying for Storage-Centric Sensor Networks". *Submitted to the ACM Transactions on Sensor Networks*, 2013

[6] S. Yang, **B. Wang** and J. S. Baras. "Interactive Tree Decomposition Tool for Reducing System Analysis Complexity". *In the 11th Annual Conference on Systems Engineering Research*, 2013.

[7] **B. Wang** and J. S. Baras. "HybridStore: An Efficient Data Management System for Hybrid Flash-based Sensor Devices". *In the 10th European Conference on Wireless Sensor Networks (EWSN)*, 2013, Ghent, Belgium.

[8] **B. Wang** and J. S. Baras. "Minimizing Aggregation Latency under the Physical Interference Model in Wireless Sensor Networks". *In IEEE SmartGridComm*, 2012, Tainan City, Taiwan.

[9] **B. Wang** and J. S. Baras. "Integrated Modeling and Simulation Framework for Wireless Sensor Networks". *In IEEE WETICE (CoMetS track)*, 2012, Toulouse, France.

[10] **B. Wang** and J. S. Baras. "Performance Analysis of Time-Critical Peer-to-Peer Communications in IEEE 802.15.4 Networks". *In IEEE ICC*, 2011, Kyoto, Japan.

[11] K. Jain, K. Somasundaram, A. R. Chowdhury, **B. Wang** and J. S. Baras. "Study of OLSR for Real-time Media Streaming over 802.11 Wireless Networks in Software Emulation Environment". *In ICST SIMUTools*, 2011, Barcelona, Spain.

[12] **B. Wang** and X. Jia. "Reduce Data Aggregation Latency by Using Partially Overlapped Channels in Wireless Sensor Networks". *In IEEE GlobeCom*, 2009, Hawaii, USA.

[13] J. Zhang, **B. Wang** and X. Jia. "Relative-Closest Connect-First Method for Topology Control in Wireless Mesh Networks". *In IEEE GlobeCom*, 2009, Hawaii, USA.

[14] D. Li, **B. Wang** and X. Jia. "Topology Control for Throughput Optimization in Wireless Mesh Networks". *In The 4th International Conference on Mobile Ad-hoc and Sensor Networks (MSN)*, 2008, Wuhan, China.

Bibliography

[1] Nicolas Tsiftes and Adam Dunkels. A database in every sensor. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, pages 316–332, 2011.

[2] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML: Systems Modeling Language (2nd Edition)*. Morgan Kaufmann Publishers Inc., 2011.

[3] Patricia Derler, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Modeling cyber-physical systems. *Proceedings of the IEEE Special Issue on CPS*, December 2011.

[4] Timothy W. Hnat, Vijay Srinivasan, Jiakang Lu, Tamim I. Sookoor, Raymond Dawson, John Stankovic, and Kamin Whitehouse. The hitchhiker's guide to successful residential sensing deployments. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, pages 232–245, 2011.

[5] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz, and David Culler. smap: a simple measurement and actuation profile for physical information. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 197–210, 2010.

[6] Stephen Dawson-Haggerty, Steven Lanzisera, Jay Taneja, Richard Brown, and David Culler. @scale: insights from a large, long-lived appliance energy wsn. In *Proceedings of the 11th international conference on Information Processing in Sensor Networks*, IPSN '12, pages 37–48, 2012.

[7] Shah-An Yang and John S. Baras. Factor join trees for systems exploration. In *Proceedings of the 23rd International Conference on Software and Systems Engineering and their Applications*, ICSSEA '11, pages 1 – 12, 2011.

[8] Yanlei Diao, Deepak Ganesan, Gaurav Mathur, and Prashant Shenoy. Rethinking data management for storage-centric sensor networks. In *CIDR*, pages 22–31, 2007.

[9] Song Lin, Demetrios Zeinalipour-Yazti, Vana Kalogeraki, Dimitrios Gunopulos, and Walid A. Najjar. Efficient indexing data structures for flash-based sensor devices. *ACM Trans. on Storage*, 2(4):468–503, November 2006.

[10] Gaurav Mathur, Peter Desnoyers, Deepak Ganesan, and Prashant Shenoy. Ultra-low power data storage for sensor networks. In *Proceedings of the 5th international conference on Information processing in sensor networks*, IPSN '06, pages 374–381, 2006.

[11] Devesh Agrawal, Deepak Ganesan, Ramesh Sitaraman, Yanlei Diao, and Shashi Singh. Lazy-adaptive tree: an optimized index structure for flash devices. In *Proc. VLDB Endow.*, volume 2, pages 361–372. VLDB Endowment, August 2009.

[12] Dongwon Kang, Dawoon Jung, Jeong-Uk Kang, and Jin-Soo Kim. μ-tree: an ordered index structure for nand flash memory. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 144–153, 2007.

[13] Suman Nath and Phillip B. Gibbons. Online maintenance of very large random samples on flash storage. In *Proc. VLDB Endow.*, pages 970–983, 2008.

[14] Suman Nath and Aman Kansal. FlashDB: dynamic self-tuning database for nand flash. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 410–419, 2007.

[15] Shaoyi Yin, Philippe Pucheral, and Xiaofeng Meng. A sequential indexing scheme for flash-based embedded systems. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 588–599, 2009.

[16] Huan Li, Dong Liang, Lihui Xie, Gong Zhang, and Krithi Ramamritham. TL-Tree: flash-optimized storage for time-series sensing data on sensor platforms. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1565–1572, 2012.

[17] Suman Nath. Energy efficient sensor data logging with amnesic flash storage. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, IPSN '09, pages 157–168, 2009.

[18] David Chu, Amol Deshpande, Joseph M. Hellerstein, and Wei Hong. Approximate data collection in sensor networks using probabilistic models. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 48–60, 2006.

[19] Amol Deshpande, Carlos Guestrin, Samuel R. Madden, Joseph M. Hellerstein, and Wei Hong. Model-driven data acquisition in sensor networks. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 588–599, 2004.

[20] Peter Desnoyers, Deepak Ganesan, and Prashant Shenoy. TSAR: a two tier sensor storage architecture using interval skip graphs. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, SenSys '05, pages 39–50, 2005.

[21] P. Derler, E.A. Lee, and A.-S. Vincentelli. Modeling cyber-physical systems. *Proceedings of the IEEE*, 100(1):13–28, 2012.

[22] Kevin Klues, Gregory Hackmann, Octav Chipara, and Chenyang Lu. A component-based architecture for power-efficient media access control in wireless sensor networks. *SenSys*, pages 59–72, November 2007.

[23] Cheng Tien Ee, Rodrigo Fonseca, Sukun Kim, Daekyeong Moon, Arsalan Tavakoli, David Culler, Scott Shenker, and Ion Stoica. A modular network layer for sensornets. *USENIX OSDI*, 2006.

[24] Elaine Cheong, Edward A. Lee, and Yang Zhao. Viptos: a graphical development and simulation environment for tinyos-based wireless sensor networks. In *ACM SenSys*, pages 302–302, 2005.

[25] M M R Mozumdar, Francesco Gregoretti, Luciano Lavagno, Laura Vanzago, and Stefano Olivieri. A framework for modeling, simulation and automatic code generation of sensor network applications. *SECON*, 2008.

[26] Derek Riley, Emeka Eyisi, Jia Bai, Xenofon Koutsoukos, Yuan Xue, and Janos Sztipanovits. Networked control system wind tunnel (ncswt): an evaluation tool for networked multi-agent systems. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques (SIMUTools)*, pages 9–18, 2011.

[27] Ludovic Samper, Florence Maraninchi, Laurent Mounier, and Louis Mandel. Glonemo: global and accurate formal models for the analysis of ad-hoc sensor networks. *1st Intl. Conf. on Integrated Internet Ad Hoc and Sensor Networks (InterSense)*, 2006.

[28] Pruet Boonma and Junichi Suzuki. Moppet: A model-driven performance engineering framework for wireless sensor networks. *The Computer Journal*, 53(10):1674–1690, 2010.

[29] IBM. IBM Rational Rhapsody Help. `www.ibm.com/software/awdtools/rhapsody/`.

[30] Dassault Systemes. Dymola. `http://www.3ds.com/products/catia/portfolio/dymola`.

[31] IEEE 802.15 TG4. IEEE 802.15.4 Standard. `http://www.ieee802.org/15/pub/TG4.html`.

[32] ITU-R. Propagation data and prediction models for indoor radio communication systems. *ITU-R Recommendations*, 2001.

[33] Ahmad T. Al-Hammouri, Michael S. Branicky, and Vincenzo Liberatore. Co-simulation tools for networked control systems. In *Proceedings of the 11th international conference on Hybrid Systems: Computation and Control (HSCC)*, pages 16–29, 2008.

[34] Baobing Wang and John S. Baras. Integrated modeling and simulation framework for wireless sensor networks. In *IEEE 21st WETICE*, pages 1 – 6, 2012.

[35] MAP FMI. Functional Mock-up Interface. `https://www.fmi-standard.org/`.

[36] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K. Arzen. How does control timing affect performance? analysis and simulation of timing using jitterbug and truetime. *IEEE Control Systems*, 23(3):16–30, 2003.

[37] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: accurate and scalable simulation of entire tinyos applications. In *ACM SenSys*, pages 126–137, 2003.

[38] Ahmad T. Al-Hammouri. A comprehensive co-simulation platform for cyber-physical systems. *Computer Communications*, 36(1):8–19, December 2012.

[39] T. Kohtamaki, M. Pohjola, J. Brand, and L.M. Eriksson. PiccSIM toolchain - design, simulation and automatic implementation of wireless networked control systems. In *Networking, Sensing and Control, 2009. ICNSC '09. International Conference on*, pages 49–54, 2009.

[40] O. Heimlich, R. Sailer, and L. Budzisz. NMLab: A co-simulation framework for matlab and ns-2. In *Advances in System Simulation (SIMUL), 2010 Second International Conference on*, pages 152–157, 2010.

[41] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *ACM SIGPLAN Notices*, 35(11):93–104, November 2000.

[42] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: scalable sensor network simulation with precise timing. In *ACM/IEEE IPSN*, 2005.

[43] Modelica Association. Modelica Language Specification 3.3. `https://www.modelica.org`.

[44] Lawrence Berkeley National Laboratory. Modelica Buildings Library. `http://simulationresearch.lbl.gov/modelica/`.

[45] Christiaan J.J. Paredis, Yves Bernard, Roger M. Burkhart, Hans-Peter de Koning, Sanford Friedenthal, Peter Fritzson, Nicolas F. Rouquette, and Wladimir Schamai. An overview of the sysml-modelica transformation specification. In *INCOSE International Symposium*, 2010.

[46] Romain Fontugne, Jorge Ortiz, Nicolas Tremblay, Pierre Borgnat, Patrick Flandrin, Kensuke Fukuda, David Culler, and Hiroshi Esaki. Strip, bind, and search: a method for identifying abnormal energy consumption in buildings. In *ACM/IEEE IPSN*, pages 129–140, 2013.

[47] QTronic GmbH. FMU SDK 1.0.2. `http://www.qtronic.de/en/fmusdk.html`.

[48] JModelica.org. PyFMI 1.2.1. `http://www.jmodelica.org/page/4924`.

[49] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection tree protocol. In *ACM SenSys*, pages 1–14, 2009.

[50] HyungJune Lee, Alberto Cerpa, and Philip Levis. Improving wireless simulation through noise modeling. In *ACM/IEEE IPSN*, pages 21–30, 2007.

[51] NS-3 Consortium. ns-3 Network Simulator. `http://www.nsnam.org/`.

[52] Judea Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. 1988.

[53] Stefan Arnborg and Andrzej Proskurowski. Linear time algorithms for np-hard problems restricted to partial k-trees. *Discrete Appl. Math.*, 23(1):11–24, 1989.

[54] H. Bodlaender. Dynamic programming on graphs with bounded treewidth. *Automata, Languages and Programming*, pages 105–118, 1988.

[55] S. Yang and J. S. Baras. Factor join trees for systems exploration. In *International Conference on Software and Systems Engineering and their Applications*, pages 1–10, 2011.

[56] Marin D. Guenov. Complexity and cost effectiveness measures for systems design. In *Manufacturing Complexity Network Conference*, pages 1–13, 2002.

[57] Stephen C. Y. Lu and Nam-Pyo Suh. Complexity in design of technical systems. *CIRP Annals - Manufacturing Technology*, 58(1):157 – 160, 2009.

[58] E.M. Clarke. Compositional model checking. In *Proceedings of Fourth Annual Symposium on Logic in Computer Science*, pages 353 – 362, 1989.

[59] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem.* PhD thesis, 1992. UMI Order No. GAX92-24209.

[60] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, 1999.

[61] E.M. Clarke. Computer-aided verification. *IEEE Spectrum*, 33(6):61 – 67, 1996.

[62] Baobing Wang and J.S. Baras. Performance analysis of time-critical peer-to-peer communications in ieee 802.15.4 networks. In *IEEE ICC*, pages 1–6, 2011.

[63] Shah-An Yang, Baobing Wang, and John S. Baras. Interactive tree decomposition tool for reducing system analysis complexity. In *the 11th Annual Conference on Systems Engineering Research*, CSER '13, pages 1 – 10, 2013.

[64] S. Yang, Y. Zhou, and J. S. Baras. Compositional analysis of dynamic bayesian networks and applications to cps. In *Conference on Systems Engineering Research (CSER)*, pages 1–10, 2013.

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

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

[67] Kaushik Chakrabarti, Minos Garofalakis, Rajeev Rastogi, and Kyuseok Shim. Approximate query processing using wavelets. *The VLDB Journal*, 10(2-3):199–223, September 2001.

[68] Waylon Brunette, Rita Sodt, Rohit Chaudhri, Mayank Goel, Michael Falcone, Jaylen Van Orden, and Gaetano Borriello. Open data kit sensors: a sensor integration framework for android at the application-level. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 351–364, 2012.

[69] Unkyu Park and John Heidemann. Data muling with mobile phones for sensornets. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, SenSys '11, pages 162–175, 2011.

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

[71] Devesh Agrawal, Boduo Li, Zhao Cao, Deepak Ganesan, Yanlei Diao, and Prashant Shenoy. Exploiting the interplay between memory and flash storage in embedded sensor devices. In *16th IEEE Intl. Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 227–236, 2010.

[72] Biplob Debnath, Sudipta Sengupta, and Jin Li. SkimpyStash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, pages 25–36, 2011.

[73] I. Lazaridis and S. Mehrotra. Capturing sensor-generated time series with quality guarantees. In *Proceedings of the 19th IEEE International Conference on Data Engineering*, ICDE '03, pages 429 – 440, march 2003.

[74] Carlos Guestrin, Peter Bodik, Romain Thibaux, Mark Paskin, and Samuel Madden. Distributed regression: an efficient framework for modeling sensor

network data. In *Proceedings of the 3rd international symposium on Information processing in sensor networks*, IPSN '04, pages 1–10, 2004.

[75] Amol Deshpande and Samuel Madden. MauveDB: supporting model-based user views in database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 73–84, 2006.

[76] Song Lin, B. Arai, D. Gunopulos, and G. Das. Region sampling: Continuous adaptive sampling on sensor networks. In *Proceedings of the 24th IEEE International Conference on Data Engineering*, ICDE '08, pages 794 –803, april 2008.

[77] Liu Yu, Jianzhong Li, Hong Gao, and Xiaolin Fang. Enabling ϵ-approximate querying in sensor networks. *Proc. VLDB Endow.*, 2(1):169–180, August 2009.

[78] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2):138–163, 2005.

[79] Atmel Inc. AT45DB041B. `http://www.atmel.com/Images/doc3443.pdf`.

[80] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[81] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[82] Enrico Perla, Art Ó Catháin, Ricardo Simon Carbajo, Meriel Huggard, and Ciarán Mc Goldrick. PowerTOSSIMz: realistic energy modelling for wireless sensor network environments. In *Proc. of the 3nd ACM workshop on Performance monitoring and measurement of heterogeneous wireless and wired networks*, pages 35 – 42, 2008.

[83] L. Peter Deutsch. DEFLATE compressed data format specification. `http://tools.ietf.org/pdf/rfc1951.pdf`, 1996.

[84] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, April 1987.