UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Improving the performance of distributed simulations of wireless sensor networks

Permalink
https://escholarship.org/uc/item/0w32x21q

Author
Jin, Zhong-Yi

Publication Date
2010

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, SAN DIEGO

Improving the Performance of Distributed Simulations of Wireless Sensor Networks

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Zhong-Yi Jin

Committee in charge:

Professor Rajesh K. Gupta, Chair
Professor William Hodgkiss
Professor Ryan Kastner
Professor Curt Schurgers
Professor Tajana Simunic-Rosing

2010

Copyright
Zhong-Yi Jin, 2010
All rights reserved.

The dissertation of Zhong-Yi Jin is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____

____

Chair

University of California, San Diego

2010

DEDICATION

To my wife, parents and brother

EPIGRAPH

Isn't it a pleasure to study and practice what you have learned?

-Confucius

TABLE OF CONTENTS

Signature Page . . . iii

Dedication . . . iv

Epigraph . . . v

Table of Contents . . . vi

List of Figures . . . viii

List of Tables . . . xi

Acknowledgements . . . xii

Vita and Publications . . . xiv

Abstract of the Dissertation . . . xv

Chapter 1 Introduction . . . 1
- 1.1 Approaches to Improve Simulation Speed . . . 2
- 1.2 Parallel and Distributed Simulations of WSNs . . . 3
- 1.3 Contributions . . . 5
- 1.4 Dissertation Organization . . . 5

Chapter 2 Overview of WSN Simulators . . . 7
- 2.1 Requirements for Designing WSN simulators . . . 7
  - 2.1.1 Overview of Wireless Sensor Networks . . . 7
  - 2.1.2 Difficulties in Building WSNs . . . 10
  - 2.1.3 Design Requirements for WSN simulators . . . 10
- 2.2 WSN Simulator Designs and Architectures . . . 12
  - 2.2.1 Simulator Designs . . . 12
  - 2.2.2 Simulator Architectures . . . 16
- 2.3 Taxonomy of WSN simulators . . . 17
- 2.4 Summary . . . 23

Chapter 3 Exploiting Application Level Parallelism in Conservative Simulations 24
- 3.1 Simulation Overheads with the Conservative Approach . . . 24
- 3.2 Technique to Exploit Application Level Parallelism . . . 25
- 3.3 Algorithm . . . 27
- 3.4 Implementation . . . 31
  - 3.4.1 PolarLite Simulator . . . 31
- 3.5 Evaluation . . . 33
  - 3.5.1 Performance in one-hop networks . . . 34
  - 3.5.2 Performance in multi-hop networks . . . 38
- 3.6 Related work . . . 41

3.7 Summary 41

Chapter 4 Exploiting Radio and MAC Level Parallelism in Conservative Simulations 43
4.0.1 Technique to Exploit Radio-level Parallelism 43
4.0.2 Technique to Exploit MAC-level Parallelism 46
4.1 Implementation 47
4.2 Evaluation 50
4.2.1 Performance of Radio-level Technique 51
4.2.2 Performance of MAC-level Technique 57
4.2.3 Performance with Both Techniques 60
4.3 Related work 62
4.4 Summary 62

Chapter 5 A New Synchronization Scheme for Conservative Simulations 64
5.1 Lazy Synchronization Scheme 64
5.1.1 Limitations of AEAP Synchronization Scheme 65
5.1.2 Lazy Synchronization Algorithm 67
5.2 Implementation 71
5.3 Evaluation 71
5.3.1 Performance in One-hop WSNs 72
5.3.2 Performance in Multi-hop WSNs 75
5.4 Related Work 76
5.5 Summary 76

Chapter 6 A Framework for Evaluating the Performance of Conservative and Optimistic Simulations of WSNs 78
6.1 Introduction 78
6.2 Idea and Approach 80
6.2.1 Ideal-Trace 83
6.3 SimVal 84
6.3.1 Conservative Playback 85
6.3.2 Optimistic Playback 89
6.4 Implementation 91
6.5 Evaluation 94
6.5.1 Accuracy of SimVal 96
6.5.2 Conservative vs. Optimistic in Simulating Single-hop WSNs 101
6.5.3 Conservative vs. Optimistic in Simulating Multi-hop WSNs 104
6.6 Related Work 106
6.7 Summary 107

Chapter 7 Conclusions and Future Work 108

Bibliography 111

LIST OF FIGURES

Figure 1.1: The progress of simulating in parallel a wireless sensor network with two nodes that are in direct communication range of each other on 2 processors. 4

Figure 2.1: System structure of the Great Duck Island bird monitoring sensor network [SMP+04] 8
Figure 2.2: Telos Mote [PSC05] 9
Figure 2.3: Taxonomy of sensor network simulators 20

Figure 3.1: The progress of simulating in parallel a wireless sensor network with two duty cycled nodes that are in direct communication range of each other 26
Figure 3.2: Average number of synchronizations per node in one-hop networks during 60 seconds of simulation time 35
Figure 3.3: Percentage reductions of the average number of synchronizations per node in one-hop networks during 60 seconds of simulation time 35
Figure 3.4: Average simulation speed in one-hop networks 36
Figure 3.5: Percentage increases of average simulation speed in one-hop networks 36
Figure 3.6: Average number of synchronizations per node in one-hop networks during 60 seconds of simulation time (a zoomed in view of Figure 3.2) 38
Figure 3.7: Average number of synchronizations per node in multi-hop networks during 20 seconds of simulation time 39
Figure 3.8: Percentage reductions of the average number of synchronizations per node in multi-hop networks during 20 seconds of simulation time . . . 39
Figure 3.9: Average simulation speed in multi-hop networks 40
Figure 3.10: Percentage increases of average simulation speed in multi-hop networks 40

Figure 4.1: The progress of simulating two nodes that are in direct communication range with the radio-level speedup technique 45
Figure 4.2: The progress of simulating two nodes that are in direct communication range with the MAC-level speedup technique 47
Figure 4.3: Speed of simulating with Avrora and PolarLite running the radio-level speedup (1 sender 31 receivers, mode 3) 52
Figure 4.4: Percentage reductions of synchronizations using the radio-level speedup technique in PolarLite (1 sender 31 receivers, mode 3) 53
Figure 4.5: Speed of simulating with and without the radio-level speedup technique in PolarLite (1 sender 31 receivers, mode 3) 53
Figure 4.6: Percentage reductions of synchronizations and percentage increases of simulation speed using the radio-level speedup technique in simulating WSNs of different sizes and radio off times in PolarLite (1 packet/10 seconds, 8 processors) 55
Figure 4.7: Total number of synchronizations in simulating WSNs of different sizes and radio off times with and without the radio-level speedup technique in PolarLite (1 packet/10 seconds, 8 processors) 56

Figure 4.8: Speed of simulating large WSNs (1 packet/10 seconds, 8 processors, mode 3) . . . 56
Figure 4.9: Percentage reductions of synchronizations with MAC-level speedup on WSNs of different sizes in PolarLite (No duty cycling) . . . 58
Figure 4.10: Speed of simulating 2 WSNs with Avrora, PolarLite and PolarLite + MAC-speedup (No duty cycling) . . . 58
Figure 4.11: Speed of simulating with MAC-level speedup on WSNs using default and double sized backoff windows (No duty cycling, 8 processors) . . . 59
Figure 4.12: Speed of simulating with Avrora, PolarLite without speedups and PolarLite with both speedup techniques (8 processors) . . . 60
Figure 4.13: Percentage increases of simulation speed and percentage reductions of synchronizations with both speedup techniques in PolarLite (8 processors) . . . 61

Figure 5.1: The progress of simulating in parallel a wireless sensor network with three nodes that are in direct communication range of each other on 2 processors. . . . 66
Figure 5.2: Performance improvements of the LazySync scheme over the AEAP scheme in simulating one-hop WSNs. Senders transmit at a 250ms interval. . . . 73
Figure 5.3: Performance improvements of the LazySync scheme over the AEAP scheme in simulating one-hop WSNs. Senders transmit as fast as possible. . . . 74
Figure 5.4: Performance improvements of the LazySync scheme over the AEAP scheme in simulating multi-hop WSNs. . . . 75

Figure 6.1: The progress of simulating with the conservative approach on one CPU a wireless sensor network of two nodes that are in direct communication range of each other. . . . 81
Figure 6.2: The ideal-trace of the simulation scenario in Figure 6.1. . . . 82
Figure 6.3: The trace of simulating a wireless sensor network with two nodes that are in direct communication range of each other. . . . 86
Figure 6.4: The ideal-trace of simulating a wireless sensor network with three nodes that are in direct communication range of each other. . . . 90
Figure 6.5: Actual and estimated simulation speed with the conservative approach on different numbers of CPUs (1 hop, 3 receivers and 1 sender) . . . 96
Figure 6.6: Simulation overheads on different numbers of CPUs (1 hop, 3 receivers and 1 sender) . . . 97
Figure 6.7: Actual and estimated total number of synchronizations with the conservative approach on different numbers of CPUs (1 hop, 3 receivers and 1 sender) . . . 97
Figure 6.8: Actual and estimated total number of context switches with the conservative approach on different numbers of CPUs (1 hop, 3 receivers and 1 sender) . . . 98
Figure 6.9: Actual and estimated simulation speed with the conservative approach on different numbers of CPUs (1 hop, 12 receivers and 12 senders) . . . 99

Figure 6.10: Simulation overheads on different numbers of CPUs (1 hop, 12 receivers and 12 senders) . . . 99
Figure 6.11: The percentage difference of the actual and estimated total number of synchronizations and context switches with the conservative approach on different numbers of CPUs (1 hop, 12 receivers and 12 senders) . . 100
Figure 6.12: Estimated simulation speeds with the conservative and optimistic approaches (1 hop, 12 receivers and 12 senders) . . . 102
Figure 6.13: Estimated simulation speeds with the conservative and optimistic approaches (1 hop, 6 receivers and 18 senders) . . . 102
Figure 6.14: Simulation overheads on different number of CPUs (1 hop, 6 receivers and 18 senders) . . . 103
Figure 6.15: Estimated simulation speeds with the conservative and optimistic approaches (multi-hop, 24 forwarders and 1 sender, 500ms transmission interval) . . . 105
Figure 6.16: Estimated simulation speeds with the conservative and optimistic approaches (multi-hop, 24 forwarders and 1 sender, 1000ms transmission interval) . . . 105

LIST OF TABLES

Table 2.1: Telos Mote Hardware Specification 9
Table 2.2: Evaluations of Sensor Network Simulators 22

Table 4.1: Radio off periods under different duty cycling modes of B-MAC 50

Table 6.1: Trace events 92
Table 6.2: Supported conservative overheads 92
Table 6.3: Supported optimistic overheads 93

ACKNOWLEDGEMENTS

First of all, I want to thank my advisor, Professor Rajesh Gupta, for his guidance, support and trust. Without him, I could never come this far and still be passionate about research. I would also like to thank Professors William Hodgkiss, Ryan Kastner, Curt Schurgers and Tajana Simunic-Rosing for serving on my Ph.D. committee. Their feedbacks and suggestions were invaluable. Special thanks go to Dr. Douglas A. Palmer for his inspiration and encouragement.

I am also grateful to my fellow students at the MESL lab: Muhammad Abdullah Adnan, Yuvraj Agarwal, Joel Coburn, Arup De, Frederic Doucet, Dohyung Kim, Sudipta Kundu, Kaisen Lin, Zhen Ma, Jeffrey Namkung, Cristiano L Pereira, Ryo Sugihara and Thomas Weng. Their support, encouragement and friendship are important parts of this work.

I would like to thank my friends William Chang, Felix Chow, Aaron Jow, XiaoJie Ma, Dr. Eric Chi-Wang Yu and many others. Our trips to the national parks, hikes in the Anza Borrego Desert, and sailings at the Mission Bay make my six years at UCSD too ephemeral.

Finally, I would like to thank my family. I want to thank my parents for encouragement and support, my brother for believing in me, and my wife, Liwen Yu, for her unconditional love.

Papers included in this dissertation

Chapter 3, in part, has been published as "Improved Distributed Simulation of Sensor Networks Based on Sensor Node Sleep Time" by Zhong-Yi Jin and Rajesh Gupta in DCOSS 08: Proceedings of the 4th ACM/IEEE International Conference on Distributed Computing in Sensor Systems [JG08], pages 204-218. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, has been published as "Improving the speed and scalability of distributed simulations of sensor networks" by Zhong-Yi Jin and Rajesh Gupta in IPSN 09: The 8th ACM/IEEE International Conference on Information Processing in Sensor Networks [JG09a], pages 169-180. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, has been published as "A New Synchronization Scheme for Distributed Simulation of Sensor Networks" by Zhong-Yi Jin and Rajesh Gupta in

DCOSS 09: Proceedings of the 5th ACM/IEEE International Conference on Distributed Computing in Sensor Systems [JG09b], pages 103-116. The dissertation author was the primary investigator and author of this paper.

Chapter 6, in part, has been submitted for publications as “A Framework for Evaluating the Performance of Conservative and Optimistic Approaches in Simulating Sensor Networks” by Zhong-Yi Jin and Rajesh Gupta. The dissertation author was the primary investigator and author of this paper.

VITA AND PUBLICATIONS

1998	B. A. in Computer Science, Boston University, Boston
2006	M. S. in Computer Science, University of California, San Diego
2010	Ph. D. in Computer Science, University of California, San Diego

PUBLICATIONS

Zhong-Yi Jin and Rajesh Gupta, "A Framework for Evaluating the Performance of Conservative and Optimistic Approaches in Simulating Sensor Networks", *Under submission*, 2010.

Zhong-Yi Jin and Rajesh Gupta, "LazySync: A New Synchronization Scheme for Distributed Simulation of Sensor Networks", In *DCOSS '09: Proceedings of the 5th IEEE International Conference on Distributed Computing in Sensor Systems*, pages 103–116, 2009.

Zhong-Yi Jin, Curt Schurgers, and Rajesh Gupta, "A Gateway Node with Duty-cycled Radio and Processing Subsystems for Wireless Sensor Networks", In *TODAES: ACM Transactions on Design Automation of Electronic Systems*, 14(1):1–17, 2009.

Zhong-Yi Jin and Rajesh Gupta, "Improving the Speed and Scalability of Distributed Simulations of Sensor Networks", In *IPSN '09: Proceedings of the 8th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pages 169–180, 2009.

Zhong-Yi Jin and Rajesh Gupta, "RSSI Based Location-Aware PC Power Management", In *HotPower '09: Proceedings of the Workshop on Power Aware Computing and Systems*, 2009.

Zhong-Yi Jin and Rajesh Gupta, "Improved Distributed Simulation of Sensor Networks Based on Sensor Node Sleep Time", In *DCOSS '08: Proceedings of the 4th ACM/IEEE International Conference on Distributed Computing in Sensor Systems*, pages 204–218, 2008.

Zhong-Yi Jin and Rajesh Gupta, "Simulations of Wireless Sensor Networks: A survey", In Technical Report CS2008-0925, UCSD, 2008.

Zhong-Yi Jin, Curt Schurgers, and Rajesh Gupta, "An Embedded Platform With Duty-cycled Radio and Processing Subsystems for Wireless Sensor Networks", In *SAMOS '07: Proceedings of the 7th International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 421–430, 2007.

ABSTRACT OF THE DISSERTATION

Improving the Performance of Distributed Simulations of Wireless Sensor Networks

by

Zhong-Yi Jin

Doctor of Philosophy in Computer Science

University of California, San Diego, 2010

Professor Rajesh K. Gupta, Chair

Simulations are key to the design, implementation, and evaluation of wireless sensor networks (WSNs) and their applications. To meet the demands for high simulation fidelity and speed, distributed simulation techniques are increasingly being used in WSN simulators. However, existing distributed WSN simulators only provide limited speedup and scalability because of the large overheads in preserving the causality of the interactions of wireless sensor nodes during distributed simulations. In this dissertation, we examine methods to improve the performance of distributed WSN simulators by controlling the overheads related to distributed simulations and parallelizing simulations.

When building distributed simulators, "conservative" and "optimistic" are the two basic approaches for preserving causality. The former ensures causality violations never occur whereas the latter features mechanisms to recover from causality violations.

These two approaches incur different overheads and their relative performances vary over different WSNs or simulation hardware. Given that all existing distributed WSN simulators are based on the conservative approach, we study, in the first part of this dissertation, how to improve the performance of the conservative approach in simulating WSNs. We first develop three novel techniques that reduce simulation overheads by exploiting the parallelism in the physical radios, communication protocols and WSN applications. Then we propose a lazy synchronization scheme that further improves simulation performance by identifying and eliminating unnecessary synchronizations during simulations. With these techniques, we implement a fully functional distributed WSN simulator.

In the second part of this dissertation, we study the performance of the optimistic approach in simulating WSNs. Our focus is on understanding the relative performance of the two approaches so appropriate simulation strategies can be devised for a WSN. Since events are handled fundamentally differently across these two classes of simulators, it is difficult to compare the approaches for a specific WSN. We address this challenge by developing a novel trace-based performance evaluation technique that separates simulation overheads from actual simulation algorithms or implementations. This allows one to use the same traces to prototype and evaluate any simulation techniques on virtual platforms with arbitrary hardware. We implement this technique in a simulation performance evaluation framework.

Chapter 1

Introduction

A wireless sensor network (WSN) is an ad-hoc network of wireless sensor nodes which are small embedded devices with on-board sensors, processors and wireless radios. Over the past decade, we have witnessed growing applications of WSNs in areas as diverse as environmental monitoring, military target tracking, and power management [SOP+04, WALJ+06, SML+04, JG09b, JG09c]. The popularity of WSNs arises from two of their unique capabilities. First, tiny sensor nodes can be deployed throughout a physical space non-intrusively, providing dense in-situ sensing of the physical phenomena that might be difficult or costly to observe previously. Second, by forming ad-hoc networks, sensor nodes are able to process and communicate this information cooperatively and autonomously without human oversight. Combining these capabilities with the Internet makes it possible to instrument the world with increasing fidelity.

Although the number of WSN applications is rapidly increasing, building WSNs remains a challenging task. We can attribute this difficulty to two main reasons. First of all, a WSN is a distributed system with a large number of sensor nodes collaborating and interacting with each other. Developing and debugging software for such a complex system on platforms with limited energy supply, computational resources, and communication capabilities are difficult by nature. Second, it is difficult to test and evaluate sensor network applications before actual deployments. Not only is it expensive to build and maintain testbeds for large sensor networks, but it is also not practical or even possible to duplicate the deployment environments in controlled settings.

Given the challenges of building WSNs to meet application requirements, simulations are key to the design, implementation, evaluation, and debugging of WSNs

and their applications [LLWC03, PBM+04, SHrC+04, TLP05, WWM07, LAW08, JG08, JG09a, JG09b]. Many other techniques have also been developed to address the same challenges. They range from new operating systems [Wor05, BCD+05], special programming languages [GLvB+03] and application specific programming models [SG08]. Simulation is particularly important among all these techniques because it provides the foundation to attack many of the problems at once and is a complement to the other techniques. By running sensor network programs on top of simulated sensor nodes inside simulated environments, the states and interactions of sensor network programs can be inspected and studied easily and repeatedly. In addition, the properties of the simulated entities (simulation models) such as the locations of sensor nodes and the inputs to the sensor nodes can be readily changed before or during simulations. By choosing appropriate simulation models, one can build an entire WSN application, including the underlying operating system, using simulations.

Compared to network level simulators such as *ns-2* [NS2], a modern WSN simulator needs to simulate more than just the network protocols. For instance, it needs to emulate the applications or operating systems to satisfy debugging needs and at the same time accurately model the hardware so low level details such as node energy usage can be simulated. These additional requirements for simulating WSNs demand the use of high fidelity simulation models [LLWC03, SHrC+04, PBM+04, LAW08]. In event driven simulations, fidelity represents the bit and temporal accuracy of events and actions. Because of the need to process a large number of events, high simulation fidelity often leads to slow simulation speed [LLWC03, JG08] which is defined as the ratio of *simulation time* to *wallclock time*. Simulation time is the virtual clock time in the simulated models [Fuj99a] while wallclock time corresponds to the actual physical time used in running the simulation program. A simulation speed of 1 indicates that the simulated sensor nodes advance at the same rate as real sensor nodes and this type of simulation is called real time simulation.

1.1 Approaches to Improve Simulation Speed

A commonly used approach to improve simulation speed is developing efficient modeling techniques that sacrifice a small degree of simulation fidelity for a large increase of simulation speed [LLWC03, LAW08]. For example, it is very computationally expensive to emulate an actual sensor node processor in a simulation model for cycle accurate

simulations [PBM+04] of WSNs. To reduce the large computational need, TimeTossim [LAW08] automatically instruments applications at source code level with cycle counts and compiles the instrumented code into the native instructions of simulation computers for fast executions. This achieves a cycle accuracy of up to 99% with only a 10 times increase of simulation time.

However, the approach of trading simulation fidelity for simulation speed has limitations. Besides reducing simulation fidelity, the biggest limitation of this approach in simulating WSNs is that it does not scale as the number of sensor nodes increases because of the increased complexity in the simulations. A scalable and widely used alternative to this approach is parallel and distributed simulations as described below [RAF+04, TLP05, WWM07, Hen08].

1.2 Parallel and Distributed Simulations of WSNs

WSN simulators can be broadly divided into two types: sequential simulators [LLWC03, LAW08] and distributed (parallel) simulators [RAF+04, TLP05, WWM07, Hen08]. Unlike sequential simulators that process all the events of a WSN in sequence on a single processor, distributed simulators can process the events of different nodes in parallel on a multi-core processor or on multiple processors and therefore can significantly improve simulation speed and scalability.

However, existing distributed WSN simulators only provide limited speedup and scalability because of the large overheads in preserving the temporal relations (causality) of the interactions of wireless sensor nodes during distributed simulations [WWM07, JG08]. When sensor nodes are simulated in parallel, their simulation speeds may vary. The variation of simulation speeds can be caused by the differences in simulated nodes or by the simulation environment. For example, different sensor nodes may run different programs or have different inputs. Also, there may not be enough processors to simulate all the nodes at the same time. Since nodes may get simulated at different speeds, it becomes critical to preserve the causality of events for correct simulations [TLP05, WWM07, JG08]. For example, as shown in Figure 1.1, two nodes in direct communication range of each other are simulated in parallel on two processors. After T_{W0} seconds of simulation, Node B is simulated faster than Node A as indicated by the fact that the simulation time of Node B (T_{S1}) is greater than the simulation time of Node A at T_{W0}. At T_{S1}, Node B is supposed to read the wireless channel and see if there is

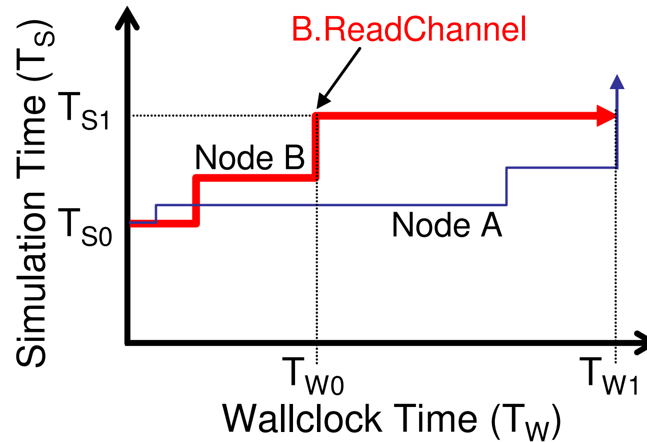

Figure 1.1: The progress of simulating in parallel a wireless sensor network with two nodes that are in direct communication range of each other on 2 processors.

an incoming transmission from Node A. However, after reaching T_{S1} at T_{W0}, Node B cannot advance any further because at T_{W0} Node B does not know whether Node A is going to transmit at T_{S1} or not. In other words, there exists in this simulation a causal relationship between the input event of Node B and the output event of Node A at T_{S1}.

There are two general approaches to preserve causality like this in distributed simulations: "conservative" [CM81] and "optimistic" [Jef85]. The conservative approach seeks to preserve causality by advancing local simulation time to the extent that the simulation is provably correct and guaranteed to be free of causality violations. For instance, with the conservative approach, Node B in Figure 1.1 has to wait at T_{S1} until Node A reaches T_{S1}.

The optimistic approach works by advancing local simulation time under well justified assumptions that reduce the likelihood of causality violations, but includes mechanisms to roll back simulation state should this occur. For example, with the optimistic approach, Node B in Figure 1.1 does not wait for the output of Node A at T_{S1} but instead advances forward by guessing whether Node A will transmit. However, the entire simulation state of Node B at T_{S1} must be saved so Node B can be rolled back to the saved state if the guess is detected to be wrong when Node A reaches T_{S1}. Since there is usually a limited amount of physical memory, special mechanisms are required to periodically compute a global safe time and release the memory used by the saved states when they become safe. Since the conservative approach has to guard against the worst case scenario, it cannot fully exploit the parallelism in the simulation like the optimistic

approach does. However, the optimistic approach also introduces additional simulation overheads from state savings and rollbacks [Fuj99b]. Since these two approaches incur different overheads, their relative performances vary over different simulation scenarios or simulation hardware [Fuj99a]. To the best of our knowledge, existing distributed WSN simulators are based on the conservative approach because it is simpler to implement and has a lower memory footprint. How well the optimistic approach will perform in simulating WSNs is an open question and we seek to address that in this dissertation.

1.3 Contributions

In this dissertation, we examine methods to improve the performance of distributed WSN simulators by parallelizing simulations and controlling the overheads related to distributed simulations. The first contribution of this dissertation is that we propose four different techniques to systematically improve the performance of the conservative approach in simulating WSNs. Using these techniques, we develop the *PolarLite* WSN simulator that provides better simulation performance in terms of simulation speed and scalability than any other existing WSN simulators of similar accuracy.

The second contribution of this dissertation is the development of a trace-based simulation performance evaluation technique for WSNs. The technique enables the prototyping and evaluation of any simulation approaches or optimization techniques without their actual implementations in real simulators. The technique also makes it possible to evaluate simulation performance on virtual platforms with arbitrary hardware. We implement this technique in the *SimVal* simulation performance evaluation framework. Using this framework, we study the relative performance of the conservative and optimistic approaches in simulating WSNs under various conditions. Based on the results of the study, we propose a new direction for future WSN simulator design.

1.4 Dissertation Organization

In Chapter 2, we explore the design requirements for a modern WSN simulator and survey the state of the art simulation techniques. Based on the survey, we classify existing WSN simulators according to their architectures, designs and levels of abstraction.

Given that existing distributed WSN simulators use the conservative approach,

in Chapters 3, 4 and 5, we investigate how to improve the performance of the conservative approach in simulating WSNs. We first develop three novel techniques that improve simulation performance by exploiting the parallelism in the WSN applications (Chapter 3), physical radios, and communication protocols (Chapter 4). Then we propose a lazy synchronization scheme (Chapter 5) that further improves simulation performance by identifying and eliminating unnecessary synchronizations during simulations. We validate these techniques by their implementations in PolarLite, a fully functional distributed cycle accurate simulator that we develop based on the Avrora simulator [JG08]. PolatLite is presented in Chapter 3.

In Chapter 6, we study the performance of the optimistic approach in simulating WSNs. Our focus is on understanding the relative performance of the two approaches so appropriate simulation strategies can be derived for a WSN. Since events are handled fundamentally differently across these two classes of simulators and there exist a large number of design tradeoffs, potential speedup techniques and optimizations for each of the approaches, it is difficult to compare the approaches for a specific WSN. We address this challenge by developing a novel trace based performance evaluation technique that separates simulation overheads from actual simulation algorithms or implementations. This allows one to use the same traces to prototype and evaluate any simulation approaches or techniques on virtual platforms with arbitrary hardware. We implement the performance evaluation technique in the SimVal simulation evaluation framework and use it to evaluate and compare the performance of the two approaches in simulating WSNs.

In Chapter 7, we conclude the dissertation and discuss future work.

Chapter 2

Overview of WSN Simulators

Simulators are important tools for the design, implementation and evaluation of WSNs and have been the subject of intense research in the past decade [LLWC03, PBM+04, SHrC+04, TLP05, WWM07, LAW08, JG08, JG09a, JG09b]. The resource constrained nature of sensor nodes, distributed operations of sensor network applications, as well as the unique working environments and deployment scenarios of sensor networks give special requirements and challenges to simulator design. More than 20 simulators have been developed or augmented for sensor networks by various companies and research groups to address different problems and needs. In this chapter, we first explore the requirements for designing a modern WSN simulator. Then we study simulator designs and architectures. In the end, we classify existing WSN simulators according to their designs, architectures and levels of abstraction.

2.1 Requirements for Designing WSN simulators

The design requirements for a modern WSN simulator are rooted in the need to address the difficulties in building WSNs and their applications. Before going into the details of the design requirements, we first give an overview of the general designs and architectures of WSNs and discuss the challenges in building WSNs.

2.1.1 Overview of Wireless Sensor Networks

We start with a real world example of a typical WSN. Figure 2.1 shows one of the earliest applications of sensor networks: habitat monitoring. In the habitat monitoring

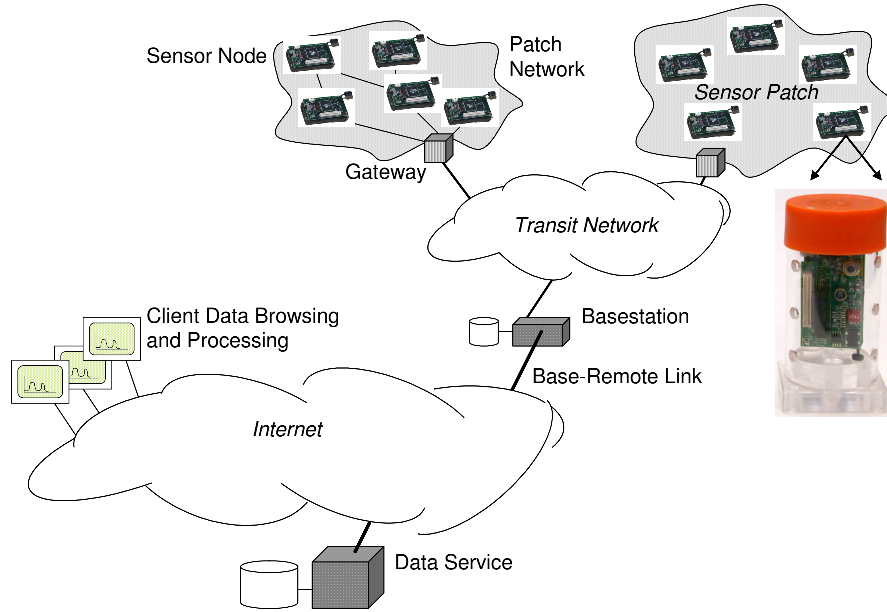

Figure 2.1: System structure of the Great Duck Island bird monitoring sensor network [SMP$^+$04]

project [SMP$^+$04], researchers focus on studying the distribution and abundance of sea birds in an offshore breading colony on the Great Duck Island, Maine. Specifically, they need to measure the occupancy of small, underground nesting burrows, and investigate the role of micro-climatic factors in habitat selection. To accomplish this, sensor nodes equipped with passive infrared (PIR), temperature and humidity sensors are placed into the burrows. PIR sensors could directly measure heat from seabirds while temperature and humidity sensors could measure variations in ambient conditions as results of prolonged occupancy. Sensor readings are collected, filtered and processed by individual nodes and transmitted via gateway nodes to the Internet. Sensor nodes automatically form patch networks so data can be relayed by other sensor nodes even if the sending nodes are not in direct communication range with the gateway nodes. Gateway nodes are responsible for bridging data between sensor networks and servers/devices in other networks. Without gateway nodes, data collected by a sensor network are only local to that sensor network and therefore no remote monitoring or interactions can be achieved [JSG09].

The sensor nodes deployed on the Great Duck Island are the Berkeley Mica Motes [HC02]. Motes are a family of sensor nodes developed by UC Berkeley and the latest in this line is the Telos Mote [PSC05] as shown in Figure 2.2. The hardware specification

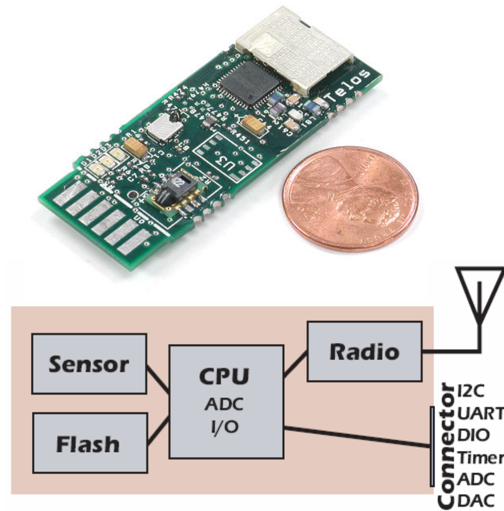

Figure 2.2: Telos Mote [PSC05]

Table 2.1: Telos Mote Hardware Specification

Sensors	Humidity, Temperature, Light
Processor	TI-MSP430, 16-bit 8MHz RISC, 3mW active, 15μW sleep, 6μS wakeup time
Memory	10KB Data, 48KB Program, 1024KB Flash
Radio	CC2420, 250kbps, 35 mW Transmit, 38mW Receive, 0.58mS turn on time, 50M indoor, 125M outdoor
Power	Two AA Batteries

of Telos Mote is listed in Table 2.1 and we can see that a sensor node typically has very limited processing, storage and communication capabilities compared to a typical desktop computer. This design is key to the success of WSNs because it keeps sensor nodes small for non-intrusive deployments, lowers the cost of sensor nodes for large scale deployments, and keeps the power consumption of sensor nodes low for long term operations. Low power is critical for WSNs because reliable energy sources often are not easily available in the deployment fields and consequently sensor nodes may have to rely on limited energy supply from batteries or harvested energy from intermittent sources like solar or wind. To ensure long lifetimes demanded by the application and deployment scenarios, sensor nodes have to be very energy efficient.

2.1.2 Difficulties in Building WSNs

From the Great Duck Island deployment example, we can see that a WSN is a distributed system with many sensor nodes collaborating and interacting with each other. Designing and developing software for such a complex system on platforms with limited energy sources, memories, processing power and communication capabilities are inherently difficult.

We can also see from the example that software for WSNs need to be highly reliable because once deployed, the costs of diagnosing and fixing the software on a large number of failed sensor nodes in the fields are extremely high. Besides, before a software program can be fixed, it may have already drained an abnormal amount of energy from the batteries, reducing the lifetimes of affected sensor nodes. However, developing reliable software for WSNs is challenging since it is difficult to debug, test and evaluate WSN applications before actual deployment. Software is difficult to debug in WSNs because sensor nodes are low cost embedded devices and usually have no displays. Once sensor nodes are detached from host computers, the only way to debug the software directly is by flashing the on-board LEDs. It is also difficult to test and evaluate sensor network applications before actual deployments because a sensor network could consist of a large number of sensor nodes. Building and maintaining a testbed of that scale is expensive. In addition, sensor network applications are driven by data collected from the physical world but it is usually not practical or even possible to duplicate the deployment environments in controlled settings.

2.1.3 Design Requirements for WSN simulators

Wireless sensor network simulators are designed to address the difficulties in building wireless sensor networks [LLWC03, PBM+04, SHrC+04, TLP05, WWM07, LAW08, JG08, JG09a, JG09b]. Similar to the integrated development environment (IDE) for software development, simulators are emerging into standard platforms for designing, developing, debugging and evaluating WSNs and their applications. By studying the challenges in building WSNs and evaluating the designs of the latest WSN simulators, we identify seven key design requirements for modern WSN simulators.

- **Sensor network specific requirements**

    1. *Real program simulation:* Use sensor node programs directly as parts of sim-

ulation models to simulate WSN applications. This requirement is driven by the complexity of programming the interactions of the sensor node applications, the operating systems and the hardware. Simply simulating the algorithms or protocols without considering low level implementation details may not reveal the true behaviors of a WSN application.

2. *Real program debugging:* Debug the running of actual sensor node programs in simulations. This complements the real program simulation requirement. Debugging should be supported at three different levels. At the application level, a simulator should provide GDB like capabilities to debug the programs on every simulated node. At the node level, a simulator should allow one to monitor and set the location and the IOs of every sensor node. For example, one should be able to monitor and change the sensor readings. At the network level, a simulator should support coordinated debugging of multiple simulated nodes. For example, set the following break condition:

```
break when ((sensor_reading_on_node_A > x &&
sensor_reading_on_node_B < y))
```

3. *Real sensor node integration:* The ability to connect simulated sensor nodes with real sensor nodes. This is important for providing simulations with real inputs such as actual sensor readings or communication patterns that are difficult to model in software.

4. *Fast prototyping:* Enable sensor network developers to quickly prototype protocols and algorithms in simulations, independent of any hardware or operating systems. By abstracting out low level implementation details, one can easily study and evaluate ideas when designing a WSN application. This requirement is a complement to the real program simulation requirement.

- **General requirements**

5. *High fidelity:* To be useful, A WSN simulator must be able to provide simulation fidelity that matches simulation needs. Fidelity indicates how accurately the behaviors (events and actions) of WSNs are simulated. In simulations, accuracy relates to both bit and temporal accuracies. For example, the accuracy of simulating a transmission event is determined by both the content of

the transmission (bit accuracy) and the time of the transmission (temporal accuracy).

6. *High performance:* A WSN simulator should provide good simulation speed and scalability [Fuj99a, LLWC03, JG08]. Simulation speed reflects how fast a scenario can be simulated. It is defined as the ratio of *simulation time* to *wallclock time*. Simulation time is the virtual clock time in the simulated models [Fuj99a] while wallclock time corresponds to the actual physical time used in running the simulation program. Scalability means a simulator should be able to simulate sensor networks that consist of large numbers of sensor nodes.

7. *High flexibility:* A WSN simulator should have a flexible design so that it can be easily extended to support different sensor network applications, sensor node operating systems, and sensor node hardware.

2.2 WSN Simulator Designs and Architectures

Most of the design requirements for WSN simulators, such as real program simulation, real program debugging, real sensor node integration, and high simulation fidelity, demand the use of highly accurate simulation models. However, simulating with these sophisticated models is computationally expensive and can significantly reduce simulation performance in terms of speed and scalability. Consequently, the fundamental design decision in building WSN simulators relates to how to provide good simulation performance while meeting other design requirements. In this section, we give a complete overview of the various design options and architectures for WSN simulators.

2.2.1 Simulator Designs

A simulator is a system that represents or emulates the behaviors of another system over time [Fuj99a]. Although it is possible for simulators to model physical systems continuously using techniques such as differential equations, WSN simulators are discrete time simulators that progress simulations by processing discrete events over time. Conceptually, a discrete time simulator can be divided into two parts: the scheduler and the simulation models. The simulation models contain the actual logics that model a physical system. The scheduler is responsible to advance the simulation time and

Listing 2.1: Main simulation loop of a discrete time-stepped simulator

```
1 for (every x seconds)
2 {
3  // advance the simulation by x seconds
4  AdvanceTimeTo(currentTime + x);
5  // simulate
6  handleSimulation();
7 }
```

drive the simulation models to produce the changes of a physical system at discrete time points. The changes of a physical system are represented as the changes of the values of some state variables in simulations. According to the ways the time points are chosen, there are two options to design a WSN simulator:

- *Time-Stepped:* Time points are divided into time steps of equal length, e.g., every clock cycle.
- *Event-Driven:* Time points correspond to simulation events, e.g., sending a packet.

The main scheduling loops for simulators designed with these two options are shown in Listings 2.1 and 2.2. For each time step or event, a handler (function) is first invoked by the scheduler and then the handler calls the right simulation models to compute the latest state of the simulated system and update the state variables if necessary. There is usually only one handler function in a time-stepped simulator but for an event-driven simulator, each event has its own handler function. Event-driven approach is usually more efficient than time-stepped one because in an event-driven simulation, evaluations of the physical system are only performed when events occur. A time-stepped simulation, on the other hand, would perform evaluations at every time step even if there are no changes in the physical system. However, without the overheads of managing events, the time-stepped design could be more efficient in cases where the changes of the simulated system only occur at some predictable time points, e.g. every clock cycle. For the same reason, time-stepped simulators are also easier to implement.

As discussed in Section 1.2, another important set of design options for WSN simulators are:

Listing 2.2: Main simulation loop of a discrete event-driven simulator

```
while(true)
{
 // if the event queue is not empty
 if(!empty(eventQueue))
 {
  // get the event with the smallest time
  // stamp (stored at the head of the queue)
  Event e = dequeue(eventQueue);
  // advance the simulation time
  AdvanceTimeTo(e.time);
  // handles the event
  Handler h = e.handler;
  h.handleEvent(e);
 }
}
```

- *Sequential:* Sensor nodes are simulated in sequence on a single processor.
- *Parallel/Distributed:* Sensor nodes are simulated in parallel on multiple processors/cores.

At a given simulation fidelity, the relative performance of the two approaches mainly depends on the amount of parallelism in the simulation, the overheads in preserving the causality and the number of processors available for the simulation [Fuj99a].

As described in Section 1.2, according to how causality is preserved, parallel and distributed simulators can be designed with the following two approaches:

- *Conservative:* Ensure causality violations never occur [CM79].
- *Optimistic:* Feature mechanisms to recover from causality violations [JBW+87].

The relative performance of the conservative and optimistic approaches in simulating WSNs is studied in Chapter 6.

Simulation Model Designs

Since simulation models contain the actual logics that model WSNs and their applications, the designs of simulation models also play an important role in meeting the design requirements for WSN simulators. There are two basic approaches to model WSN applications:

- *Emulation:* Model the processor and IO devices of a sensor node so compiled WSN programs can be executed by the models directly without modifications.
  - *Direct-execution* A special type of emulation. The simulation computers where the WSN application models are executed use exactly the same instruction set as the sensor node processor. Therefore, compiled WSN programs can be executed natively on the simulation computers.
- *Simulation:* Model a representation of the original program. The representation could be an approximation of the original program or is equivalent to the original program but in other languages or instruction sets.

Emulation enables *cycle-accurate* simulations of WSNs because an emulator executes instruction by instruction the same machine code that runs on a real sensor node

processor. Simulation is not as accurate but allows fast-prototyping because it is easier to develop WSN applications in high level modeling languages without concerning with low level operating system and hardware details.

2.2.2 Simulator Architectures

As shown in Figure 2.2, a sensor node can be physically divided into four major units: the sensing unit, the processing unit, the communication unit and the power unit. The sensing unit includes all the sensors. It interacts with the environment as well as the processing and power units. The processing unit contains one or more [JSG07] processors and their associated memories (Flash/RAM/ROM). It interfaces with the other three units. The communication unit includes the radio. It directly interacts with the local processing unit and via wireless channels the communication units on other nodes. The power unit supplies energy to all other units and may interact with the environment if it collects energy from the environment or is affected by environmental factors such as temperature.

Typically, a sensor network simulator is structured similarly to the real sensor networks that it models. It includes models that represent the four major units of a sensor node, the deployment environments and the communication channels. To simulate a WSN, we only need to configure and connect the appropriate simulation models and, if necessary, develop additional models. There are two very different approaches to connect simulation models and represent their interactions:

- Event-oriented
- Process-oriented

These two approaches correspond to how one would view the physical world. In the event-oriented view, one would regard the physical world as an event driven system. The changes of the physical world are modeled by events and handlers of these events. This view is the basis of the discrete event driven simulator design. In the process-oriented view, the world is modeled as a set of self contained and autonomous entities running in parallel and interacting with each other. The process-oriented view is more intuitive and fits naturally with object oriented programming concepts. This makes it easier to write, test, extend and maintain simulation models. However, since the process-oriented view can not be mapped directly to the event driven simulator design, a

supporting layer has to be added for simulators employing the process oriented approach and that introduces additional overheads.

To demonstrate the differences of the event-oriented and process-oriented approaches, we show in Listings 2.3 and 2.4 examples that simulate a WSN with these two approaches respectively. The simulated WSN consists of two Nodes, A and B. Node A sends a packet to Node B first. If a reply from Node B is received by Node A, Node A sends another packet to Node B 5 seconds later. The process repeats until terminated by the user. For simplicity, we only show the modeling of Node A and assume the communication channels are perfect without any packet loss.

2.3 Taxonomy of WSN simulators

According to the architectures and designs discussed in the previous two sections, the top half of Figure 2.3 classifies the following sensor network simulators and simulators that could potentially be used to simulate WSNs: ns-2 [NS2], SensorSim [PSS00, PSS01], SENSE [CBMPS04], GTSNetS [OAVRHR05], OPNET [Cha99], J-Sim [SCH+05, SCH+06], OMNet++ [Var01], SENSIM [MSK+05], TOSSIM [LLWC03], ScatterWeb [WS06], EmTOS [GSR+04], SenQ [VXSB07], TOSSF [PN02], NesCT [Nes], Viptos [CLZ06], ATEMU [PBM+04], Avrora [TLP05], DiSenS [WWM07], SimGate [WGC+06], and EmStar [GEC+04].

From a simulator user's point view, a more intuitive method to categorize sensor network simulators is by looking at the levels of physical abstraction they offer for simulations. We define three levels of abstraction as shown in the bottom half of Figure 2.3.

Listing 2.3: Code example for event-oriented approach

```
1 main() {
2   // create a new event which will be handled
3   // by NodeASendEvent_Handler
4   Event e = new NodeASendEvent(NodeASendEvent_Handler)
5   // schedule the event right away
6   ScheduleEvent(0, e);
7 }
8
9 NodeASendEvent_Handler() {
10  // send a packet to node B. This is done by
11  // creating a new packet (event) that is handled
12  // by the NodeBReceiveEvent_Handler of Node B
13  Event e;
14  e = new NodeBReceiveEvent(NodeBReceiveEvent_Handler);
15  e.content=packet;
16  // models the TX delay by scheduling the packet
17  // txDelay seconds later
18  ScheduleEvent(txDelay, e);
19 }
20
21 // handles the reply from node B
22 // Send a new packet to Node B 5 seconds later
23 NodeAReceiveEvent_Handler(NodeAReceiveEvent re) {
24  // create a new event which will be handled
25  // by NodeASendEvent_Handler
26  Event e = new NodeASendEvent(NodeASendEvent_Handler)
27  // schedule the event in 5 seconds
28  ScheduleEvent(5, e);
29 }
```

Listing 2.4: Code example for process-oriented approach

```
1 NodeA {
2   while(true)
3   {
4     // send a packet to Node B with txDelay
5     Send_To(NodeB, packet, txDelay);
6     // wait for a reply from Node B
7     Wait_Until(Receives packet from Node B);
8     // continue in 5 seconds
9     Advance_Time(5);
10  }
11 }
```

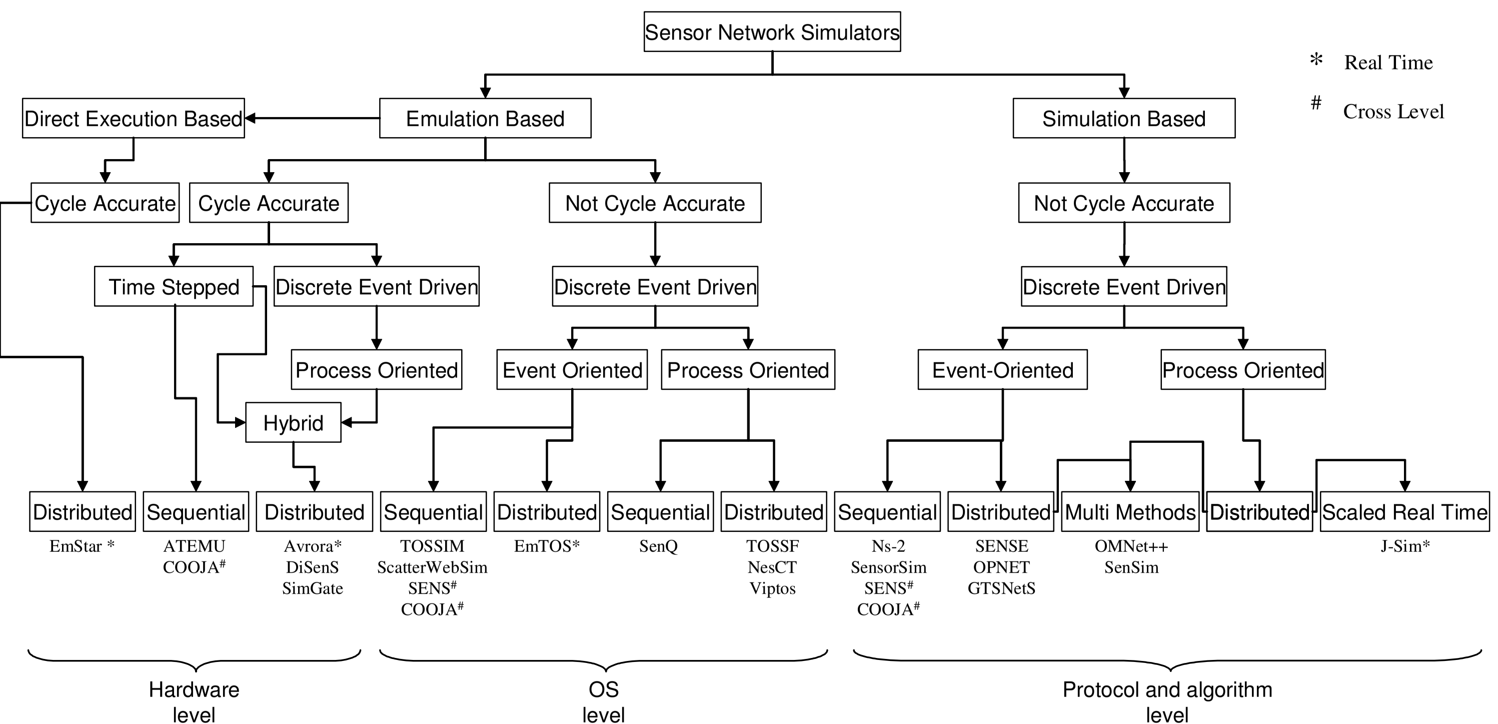

Figure 2.3: Taxonomy of sensor network simulators

At the highest abstraction level, the protocol and algorithm level, most of the low level details of the sensor node hardware and operating systems are abstracted away. This level of abstraction provides an ideal environment for rapidly prototyping and evaluating high level protocols and algorithms independent of sensor node platforms and operating systems.

We call the middle level the operating system (OS) level. This level models the services of an operating system with such details that applications written for that operating system can run on top of it without modifications. As a result, algorithms and protocols can be evaluated as parts of a real application but independent of any specific sensor node hardware.

The bottom level is the hardware level. This level models the functions of major hardware components of a sensor platform, mainly the processor and associated IO devices. Since the entire physical platform is modeled, a simulator can execute clock cycle by clock cycle, instruction by instruction the same binary program that is executed by a real sensor node. In other words, we can boot from it, without modifications, any operating systems capable of running on the corresponding physical hardware. This makes it possible to study the accurate timing and interrupt behaviors of operating systems and applications.

We can see from Figure 2.3 that the two types of classifications match well. All protocol and algorithm level simulators are simulation based while OS level simulators and hardware level simulators are emulation based. If we follow the tree in Figure 2.3 down, we can see that all OS level simulators are not cycle accurate but all hardware level ones are. The correlations arise from the fact that different architectures and designs are chosen for different levels of abstraction. A simulator does not need to be constrained to a single abstraction level. For example, COOJA [ODE+06] is a multi-level simulator that works at all three abstraction levels. In Table 2.2, we compare the features offered by simulators of different abstraction levels.

Table 2.2: Evaluations of Sensor Network Simulators

	Real program simulation	Real program debugging	Real sensor node integration	Fast prototyping	Fidelity	Performance	Flexibility
Protocol and Algorithm level	-	-	+	+	+	+++	+++
OS level	+	+	++	-	++	++	++
Hardware level	++	++	+++	-	+++	+	+

2.4 Summary

In this chapter, we gave a general overview of WSN simulators. In particular, we considered the requirements for simulating WSNs and presented in detail the designs and architectures of existing WSN simulators. We have shown that the major design decision in building WSN simulators is to provide good simulation performance while meeting other requirements. We have also developed a taxonomy that partitions WSN simulators based on their designs, architectures and levels of abstraction.

Chapter 3

Exploiting Application Level Parallelism in Conservative Simulations

As described in Chapter 1, “conservative” and “optimistic” are the two basic approaches to preserve causality in distributed simulations. Existing distributed WSN simulators are based on the conservative approach since it is simpler to implement and has a lower memory footprint [Fuj99a]. In this chapter, we present a technique that improves the performance of the conservative approach in simulating WSNs by exploiting the parallelism available in WSN applications. In particular, we seek to use the information regarding duty cycling of nodes in WSNs to speed up simulations. Before discussing the speedup technique in detail, we first summarize the types of overheads of the conservative approach in distributed simulations of WSNs.

3.1 Simulation Overheads with the Conservative Approach

As described in Section 1.2, in distributed simulations, simulated sensor nodes have to synchronize with each other to preserve causality. Synchronizations bring significant overheads to distributed simulations. With the conservative approach, the overheads can be divided into management overheads and communication overheads.

Management overheads come from managing the threads or processes that simulate sensor nodes. For example, to maximize the parallel use of computational resources,

the thread or process simulating a waiting Node, e.g., Node B in Figure 1.1 between T_{W0} and T_{W1}, needs to be suspended so another thread or process simulating a different node can be swapped in for execution. Suspended nodes also need to be swapped back in for simulation later on. These usually involve context switches and large numbers of them would significantly reduce simulation speed and scalability.

Communication overheads arise because nodes need to communicate their progresses to each other during simulations. For example, in Figure 1.1, Node A must notify Node B after it advances past T_{S1} so that Node B can continue. Communicating across processes or threads is generally expensive. In the case where nodes are simulated on different computers, the communication overheads could be very high since synchronization messages have to be sent across the physical networks and through the protocol stacks before reaching the destination nodes.

3.2 Technique to Exploit Application Level Parallelism

Sensor node synchronizations are required for enforcing dependencies between sensor nodes in simulations. Since the dependencies come from the interactions of sensor nodes over wireless channels, the number of required synchronizations in a distributed simulation is inversely proportional to the degree of parallelism in the WSN application under simulation [Fuj99a].

To reduce the number of synchronizations, we exploit the parallelism available in WSN applications. In particular, we seek to use the information regarding duty cycling of nodes in sensor networks to speed up simulations. Node duty cycling is common in WSN applications for power management purposes. Most WSN applications have very low duty cycles and need to carefully manage their power consumptions in order to function for an extended period of time under the constraint of limited energy supply. In other words, sensor nodes sleep most of the time and do not interact with each other frequently until certain events are detected [SPMC04]. Very little power is consumed by a sensor node in the sleep state since its wireless radio is turned off and its processor is put into a low power sleep mode.

Our speedup technique is illustrated in Figure 3.1 which shows the progress of simulating two duty cycled sensor nodes that are within direct communication range of each other. In the simulation, Node B enters into the sleep state at $T_{S0'}$ and wakes up at $T_{S1'}$. With existing distributed WSN simulators, Node A needs to wait for Node B at T_{S1}

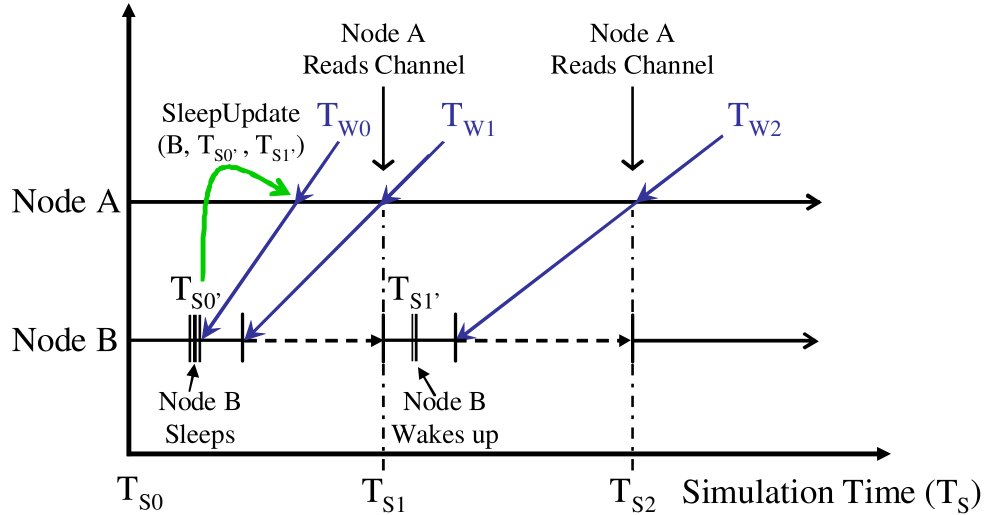

Figure 3.1: The progress of simulating in parallel a wireless sensor network with two duty cycled nodes that are in direct communication range of each other

although Node B does not transmit anything during its sleep period. To eliminate this type of unnecessary synchronization, our technique keeps track of the time that a node enters into the sleep state and the time it wakes up. When we detect during a simulation that a node is entering into the sleep state, we immediately send both the entering time and exiting time (simulation time) in a *SleepUpdate* message to the neighboring nodes that are within direct communication range. As a result, neighboring nodes no longer need to synchronize with the sleeping node during the sleep period. For example, when we detect that Node B is entering into the sleep state at $T_{S0'}$, we immediately notify Node A that Node B will be in the sleep state from $T_{S0'}$ to $T_{S1'}$. Once Node A knows that Node B will not transmit between $T_{S0'}$ and $T_{S1'}$ and $T_{S0'} \leq T_{S1} < T_{S1'}$, it no longer needs to wait for Node B at T_{S1} and its lookahead time increases. Lookahead time is defined as the amount of simulation time that a simulated sensor node can advance freely without waiting for inputs from other simulated sensor nodes [Fuj99a]. The speedup of our technique increases with the durations of sleep periods because the longer the sleep periods, the larger the lookahead time.

For the speedup technique to work, we have to be able to detect both sleep time and wakeup time in simulations. Sleep time can always be detected because a real sensor node processor has to execute some special instructions to put the node into the sleep mode. Correspondingly, sleep events, which are integral parts of any WSN

simulators, are used in simulations to signal the transitions of nodes from active states to the sleep state. For example, in the case of cycle accurate sensor network simulators, sleep events are associated with the execution of specific machine instructions of the sensor node processors under simulation. However, detecting wakeup time is a challenging process since a node can be woken up by interrupts from either a timer or an external autonomous sensor such as a passive infrared sensor. Autonomous sensors are devices that can function independently without the support of a node processor and therefore may wakeup a sensor node at any time.

The wakeup time for sensor nodes that do not have autonomous sensors can always be detected. This is because the wakeup time has to be passed to a physical timer before a real sensor node is able to enter into the sleep state since the node processor can not execute any instructions during the sleep mode. For nodes equipped with autonomous sensors, if the input events to the autonomous sensors are known before a simulation starts, which is generally the case, the wakeup time can always be computed as the smaller of the time of the timers and the input events. If the input events to the autonomous sensors are not known before a node enters into the sleep mode, for example, inputs are collected in real time from real sensors [GEC+04], then the speedup technique has to be disabled on that node. However, we only need to turn off the speedup technique on those nodes receiving unpredictable input events, while other nodes can still use the technique.

3.3 Algorithm

Before an algorithm for the proposed speedup technique can be developed, we have to first build a distributed conservative synchronization algorithm that enables the exploiting of application level parallelism for speedup. For such purposes, we develop a generic distributed conservative synchronization algorithm that is similar to the one in DiSenS [WWM07] and suitable for both parallel and distributed simulations of WSNs. On top of our synchronization algorithm, we develop the speedup algorithm to exploit the application level parallelism. It is shown together with the synchronization algorithm in Algorithm 1.

Algorithm 1: Conservative Synchronization Algorithm with Speedup Technique

Require: $nl := \{< nid, nclock >\}$ /*a list of neighboring node ids and their reported simulation time*/

Require: id /*current node ID*/, $bytetime$ /*the amount of time to transmit one byte with a wireless radio*/

1. $clock \Leftarrow 0$ /*current sim clock to 0*/, $lookahead \Leftarrow bytetime$, $intervalclock \Leftarrow 0$
2. **for** every tuple $< nid, nclock >$ in nl **do**
3. $nclock \Leftarrow 0$ /*initialize simulation time of neighboring nodes to zero before starting the simulation*/
4. **end for**
5. **while** $clock \leq$ user inputed simulation time **do**
6. $waitchannel \Leftarrow false$
7. execute *next instruction*
8. **if** the *instruction* puts a node into the sleep state **then**
9. $exitclock = wakeuptime$, $intervalclock \Leftarrow exitclock$
10. send a $ClockUpdate(id, exitclock)$ message to every nid node in the tuple $< nid, nclock >$ of nl
11. **else if** the *instruction* reads from the wireless radio **then**
12. **if** $lookahead \geq 0$ **then**
13. read the wireless radio
14. **else**
15. **if** $intervalclock \neq clock$ **then**
16. $intervalclock \Leftarrow clock$
17. send a $ClockUpdate(id, clock)$ message to every nid node in the tuple $< nid, nclock >$ of nl
18. **end if**
19. $waitchannel \Leftarrow true$
20. **end if**
21. **end if**
22. **if** $waitchannel$ is $false$ **then**
23. $clock \Leftarrow clock + cyclesconsumed$ /*advance clock by the clock cycles of the executed instruction*/
24. **end if**

25. $updated \Leftarrow false$ /*check incoming $ClockUpdate$ messages at least once per instruction*/
26. **repeat**
27. **for** each received $ClockUpdate(cid, cclock)$ message **do**
28. **for** every tuple $< nid, nclock >$ in nl **do**
29. **if** uid equals to cid **then**
30. $nclock \Leftarrow cclock$
31. **end if**
32. **end for**
33. **end for**
34. $updated \Leftarrow true$
35. $minclock = min(nclock)$ in the tuple $< nid, nclock >$ of nl /*find the neighbor with the smallest clock*/
36. $lookahead \Leftarrow (minclock - floor(clock/bytetime) * bytetime)$ /*on byte boundaries with byte-radio*/
37. **if** $lookahead \geq 0$ and $waitchannel$ is $true$ **then**
38. read the wireless radio /*all neighbors have advanced past the byte boundary this node is waiting on*/
39. $clock \Leftarrow clock + cyclesconsumed$, $waitchannel \Leftarrow false$
40. **end if**
41. **until** $waitchannel$ is $false$ and $update$ is $true$
42. **if** $(clock - intervalclock) \geq bytetime$ **then**
43. $intervalclock \Leftarrow clock$
44. send a $ClockUpdate(id, clock)$ message to every nid node in the tuple $< nid, nclock >$ of nl
45. **end if**
46. **end while**

Synchronizations are only necessary between neighboring nodes that are within direct communication range of each other. The first step before applying our algorithm is to build a neighbor node list for each node according to the locations of the sensor nodes and the maximum transmission range of their wireless radios. Mobile nodes need to be included in the neighbor node list of all other nodes. Then, a time stamp is assigned to every node (node id) in the lists to keep the last reported simulation time of that node.

This list, named *nl* in Algorithm 1 is the first required input to the synchronization algorithm. There are two more inputs to the algorithm. The second input *id* is used to identify the node under simulation. The nodes in *nl* are neighbors of this node. The third input *bytetime* is the amount of time to transmit one byte with a wireless radio. It is the maximum lookahead time without synchronizations. Every node starts with that lookahead time because it takes that amount of time for one byte of data to travel from the sender to the receiver. For example, if a node starts at simulation time 0 and wants to read the wireless channel at that time, it can do so because the earliest time that a byte of data can arrive is $0 + bytetime$. Similarly, after synchronizing at time T_S, all synchronized nodes can advance freely up to $T_S + bytetime$ without any additional synchronizations. However, this approach only works if the processor and radio on a real sensor node communicate by exchanging data one byte at a time (*byte-level*).

The variable *intervalclock* in Algorithm 1 is used to ensure that the *ClockUpdate* messages are sent by every simulated node once every *bytetime* if the node is not already in the sleep state. These messages update neighboring nodes about the latest simulation time of the sender and ensure neighboring nodes have the right time information to make synchronization decisions according to Condition 1. The interval chosen to send the messages will affect the performance of the algorithm as nodes may have to wait if *ClockUpdate* messages are delayed. The smallest interval one can use with *byte-level* radios is *bytetime* because the actual waiting time must fall on byte boundaries. This can be seen in Algorithm 1 where the *floor* function is used to calculate the lookahead time.

Condition 1 *If a node N_i reads data sent by a node N_s over a wireless channel C_k at simulation time T_{SN_i}, then the simulation time of node N_s, T_{SN_s}, must be greater than or equal to T_{SN_i}.*

The *SleepUpdate* message described in Section 3.2 is replaced in Algorithm 1 with the *ClockUpdate* message because receiving nodes only need to use the wakeup time of the sender to calculate lookahead time. However, to take advantage of a special optimization described in the future work part of Section 3.7, *SleepUpdate* messages must be used so the time that a node enters into the sleep state is sent as well. The time that a node enters into the sleep state is detected when the *Sleep* instruction of the ATMega128L microcontroller [Atm03] is executed. This instruction can put the microcontroller into different sleep modes based on the values set in the *MCU Control*

Register. It is critical to read that register before sending any *ClockUpdate* messages as the speedup technique only works if a microcontroller is put into *Power-Save Mode*. The reason is that the only way to wake up a microcontroller from *Power-Save Mode* is through interrupts generated by timers or external autonomous sensors. Because of that, we can find the wakeup time by keeping track of the values written to the *Timer Control Register* and using the techniques described in section 3.2.

3.4 Implementation

To accurately evaluate the performance of the speedup technique, we develop the PolarLite simulator.

3.4.1 PolarLite Simulator

PolarLite is a distributed conservative WSN simulator that we develop on top of the Avrora simulator [TLP05]. It uses the same simulation models as the ones in Avrora but runs our distributed synchronization algorithm in Algorithm 1. The lock-step style synchronization algorithm of Avrora is optimized for parallel simulations on SMP computers but lacks necessary features to support our speedup technique.

We choose to build PolarLite on top of Avrora because Avrora is a widely used cycle accurate sensor network simulator. Among all types of sensor network simulators, cycle accurate sensor network simulators [PBM+04, TLP05, WWM07] offer the highest level of fidelity. They provide simulation models that emulate the functions of major hardware components of a sensor node, mainly the processor. Therefore, one can run on top of them, clock cycle by clock cycle, instruction by instruction, the same binary code (images) that are executed by real sensor nodes. As a result, accurate timing and interactive behaviors of sensor network applications can be studied in details.

Avrora is written in Java and supports parallel simulations of sensor networks comprised of Mica2 Motes [Cro08]. It allocates one thread for each simulated node and relies on the Java virtual machine (VM) to assign runnable threads to any available processors on an SMP computer. Our implementation of PolarLite is based on the Beta 1.6.0 code release of Avrora which is well documented and publicly available. Implementing PolarLite on top of Avrora mainly involves developing new code in two areas: channel modeling and synchronization algorithm.

Channel modeling

With our synchronization algorithm, a node can write to a wireless channel long before the packets are read by other nodes (the transmitting code is omitted in Algorithm 1 for simplicity). Because of this, we develop a new wireless channel model that uses a circular buffer to store unread wireless packets. Our channel model uses a similar method as the original channel model in Avrora to map transmitting and receiving time into time slots that are *bytetime* apart. The slot number is used to index into the circular buffer. Note that with the original channel model, a write to a channel right after synchronizations could be dropped as the write may happen before the time slots are carried forward. Our channel implementation does not drop data.

Complexity of Synchronization Algorithm

The computational complexity of a synchronization algorithm is determined by the total number of synchronization messages that need to be sent and the overheads in sending and processing each of the messages [Nic98]. Our synchronization algorithm has higher computational complexity than the one in Avrora because our algorithm is designed as an unoptimized generic distributed algorithm. In Avrora, a global data structure is used to keep track of the simulation time of each node and therefore a node only needs to update the global data structure once for each clock update. This centralized approach is optimized for parallel simulation over SMP computers but does not support distributed simulations over a network of computers. We implement our synchronization algorithm as a truly distributed algorithm by distributing parts of the global data structure to each node. The penalty is that a node with N nodes in direct communication range has to send a total of N messages for each clock update. However, the penalty is not significant when the number of nodes within direct communication range is not big. In fact, because the synchronization algorithm of Avora works by synchronizing a node with all other nodes regardless of whether they are within communication range or not, our distributed algorithm may even perform better when nodes are sparsely distributed. If performance is an issue in the future, we could choose to optimize our implementation for parallel simulations using a centralized approach.

3.5 Evaluation

We conduct a series of experiments to evaluate the performance of our speedup technique. The experiments are conducted on an SMP server running Linux 2.6.9. The server has 4 processors (Intel Xeon 2.8GHz) and 2GByte of RAM. For comparison, the same test cases are simulated using both our modified Avrora and the original Avrora. Sun's Java 1.6.0 is used to run both simulators.

To demonstrate the effectiveness of our speedup technique, we choose two programs from the CountSleepRadio example which is a part of the TinyOS 1.1 distribution [HSW+00, TA]. These two programs behave exactly like the CntToRfm and RfmToLeds programs used in the experiments of the Avrora paper and serve similar purposes. The only difference is that the new programs can put sensor nodes into sleep states to save power. (The latest TinyOS 2.0 release [LGH+05] is not used for our experiments because its radio stack is not fully compatible with the radio model in the version of the Avrora that our code is based on.)

The first program we use for our experiments is CountSleepRadio. It wakes up a node periodically from the sleep state to increase the value of a counter by one and broadcast that value in a packet. Once the packet is sent, it puts the node back into the sleep state to save power. We have to modify this program for some of our experiments because the original program has an upper bound on how long a sensor node can stay in the sleep state [1]. This is unnecessary and we work around this limitation by using the *Clock* interface of the TinyOS 1.1 directly. The problem has reportedly been fixed in TinyOS 2.0. The counterpart of our CountSleepRadio program is CountReceive. It receives packets sent by CountSleepRadio and flashes different LEDs based on the values in the packets. For simplicity, we identify nodes running CountSleepRadio and CountReceive as senders and receivers respectively in the following sections.

Both CountSleepRadio and CountReceive use the default TinyOS 1.1 CC1000 CSMA (carrier sense multiple access) MAC (media access control) which is based on B-MAC [PHC04]. Before sending a packet, the CC1000 MAC first backs off for a random amount of time and then reads its transmitting channel for ongoing transmissions. It only sends the packet if the channel is clear. Otherwise, it backs off for a random amount of time before checking the channel again. As a result, a sender in our experiments reads

[1] The upper bound is imposed by the timer implementation of TinyOS 1.1. The timer code sets $maxTimerInteval$ to $230ms$ and the physical timers of a real sensor node can not be set to anything larger than that using the timer API.

the wireless channel at least once before each transmission.

3.5.1 Performance in one-hop networks

In this section, the performance of our speedup technique is evaluated under various sleep times and network sizes using one-hop sensor networks. One-hop sensor networks are sensor networks set up in such a way that all sensor nodes are within direct communication range of each other. It is a common form of sensor network used in actual deployments [SPMC04].

In the one-hop sensor network experiments, nodes are laid on a 10 by 10 grid 1 meter apart and their maximum transmission ranges are set to 20 meters. A fixed node is selected as a receiver and the rest as senders. The receiver listens continuously like a gateway node [SPMC04, JSG07] and does not enter into the sleep state. The senders are duty cycled and their sleep durations are varied for different experiments. Sleep duration is how long a node stays in the sleep state before waking up. All results in this section are averages of three runs.

Figure 3.2 shows the average number of synchronizations per node in one-hop networks during 60 seconds of simulation time. Since all nodes are simulated for the same number of clock cycles (60 $\times$ clock frequency of ATMega128L) in all test cases, the average number of synchronizations per node is a good indicator to the performance of the speedup technique. The synchronization numbers are collected by logging code we add specifically for evaluation purposes. Figure 3.3 shows the percentage reductions of the average number of synchronizations per node in one-hop networks during the 60 seconds of simulation time. Figure 3.4 shows average simulation speed in one-hop networks. The average simulation speed V_{avg} is calculated using Equation 3.1. The percentage increases of average simulation speed in one-hop networks are shown in Figure 3.5.

$$V_{avg} = \frac{total\ number\ of\ clock\ cycles\ executed\ by\ the\ sensor\ nodes}{(execution\ time\ of\ the\ simulation) \times (number\ of\ sensor\ nodes)} \tag{3.1}$$

As shown in Figure 3.2 and Figure 3.3, the speedup technique significantly reduces synchronizations in all the test cases and the largest percentage reduction is more than 99%. The reduction percentages increase with sleep durations under fixed network sizes except for the 16 (1 receiver, 15 senders) and 32 (1 receiver and 31 sender) node

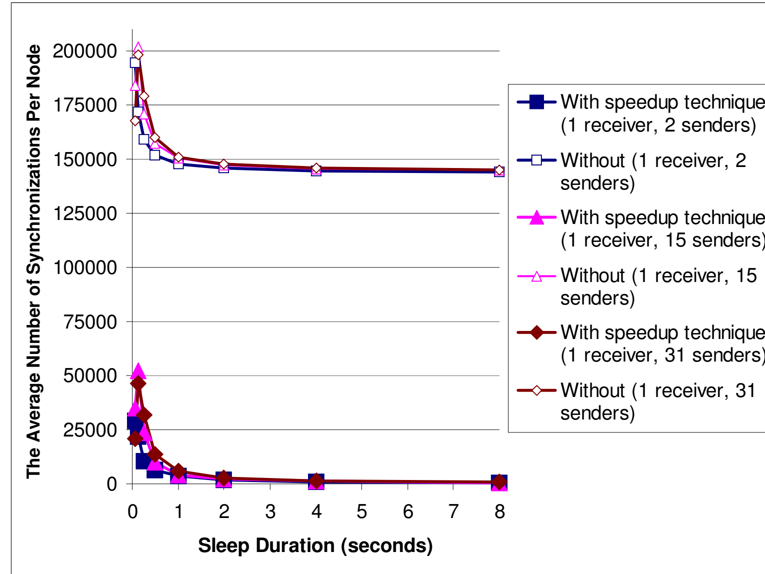

Figure 3.2: Average number of synchronizations per node in one-hop networks during 60 seconds of simulation time

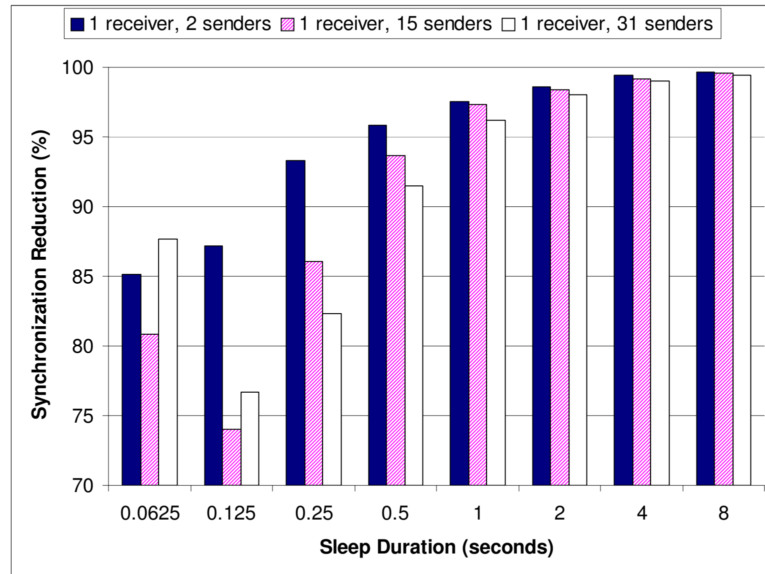

Figure 3.3: Percentage reductions of the average number of synchronizations per node in one-hop networks during 60 seconds of simulation time

test cases with 62.5ms sleep durations. The unusually high percentage reductions in those cases are results of using the CC1000 CSMA MAC protocol of TinyOS 1.1. When multiple senders in communication range transmit at the same time, the MAC protocol

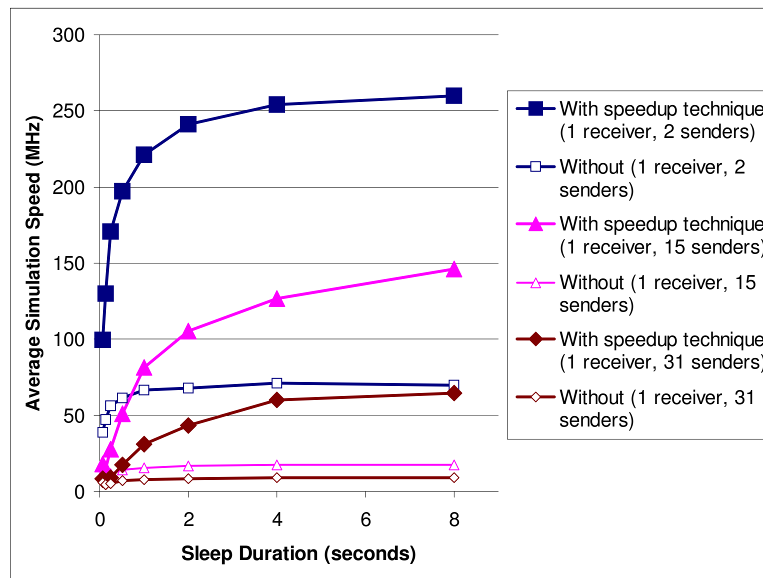

Figure 3.4: Average simulation speed in one-hop networks

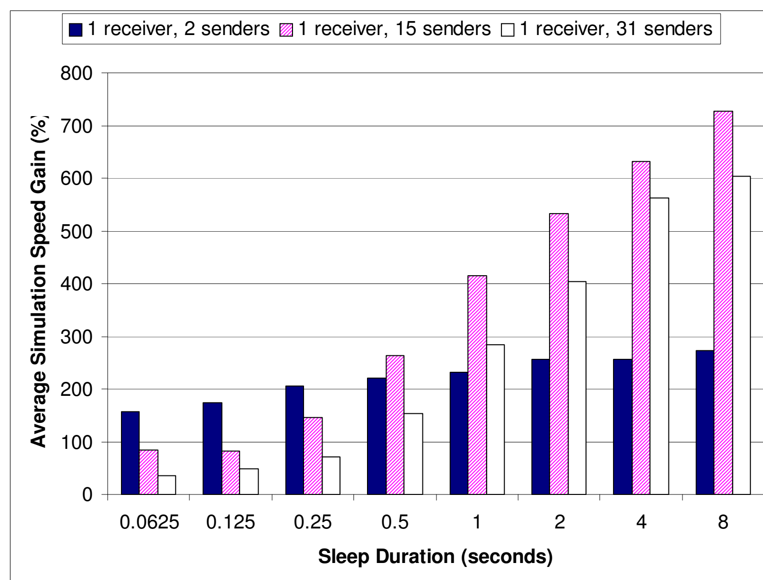

Figure 3.5: Percentage increases of average simulation speed in one-hop networks

would sequence their transmission times using random backoffs. Since the senders will not return back to the sleep state until packets are successfully transmitted, the sleep times of the senders are sequenced as well. This effectively reduces synchronizations in simulations as the number of nodes that are active at a same time is reduced. The

3-node (1 receiver, 2 senders) test case is not affected by this because we randomly delay the starting time of each node between 0 and 1 second in all our experiments to prevent the nodes from artificially starting at the same time. When the number of nodes in a one-hop network decreases, the chance for concurrent transmissions decreases and the number of synchronizations increases. Similarly, the chance for concurrent transmissions also decreases when the sleep duration increases because the nodes in a one-hop network would transmit less frequently with larger sleep durations. As a result, the number of synchronizations increases with sleep durations in such cases. We can see this from Figure 3.6, a zoomed in view of Figure 3.2. When sleep duration doubles from $62.5ms$ to $125ms$, the average number of synchronizations per node actually increases for both of the 16 and 32 node test cases, regardless of whether the speedup technique is used or not. We can also see in the same test cases that the average number of synchronizations per node decreases with network size under fixed sleep durations. The speedup technique can further reduce synchronizations in cases like these because it increases the lookahead time of simulated nodes. The reduction from applying the speedup technique is greater for larger one-hop networks in those cases as there is more of this type of sequencing in larger one-hop networks under fixed sleep durations.

As shown in Figure 3.4 and Figure 3.5, the speedup technique significantly increases average simulation speed in all the test cases and the largest increase is more than 700%. Although the 3-node speedup test case has the highest percentage reduction of the average number synchronizations per node as shown in Figure 3.3, it does not have the largest average simulation speed increase in Figure 3.5. This is because the overhead in performing a synchronization is very low for the 3-node test cases. Context switches are generally not needed for synchronizations in those cases because there are more processors (4) than nodes/threads (3). We can also see in Figure 3.5 that the growth of the average simulation speed quickly flattens out for large sleep durations in original Avrora but continues after applying the speedup technique. We expect even better average simulation speed in simulating large one-hop networks after optimizing our generic distributed synchronization algorithm for parallel simulations as discussed in Section 3.3.

The speedup technique can not completely eliminate the synchronizations caused by having only limited numbers of physical processors available for simulations. We can see this from Figure 3.2 and Figure 3.6. When the sleep duration is long enough, the

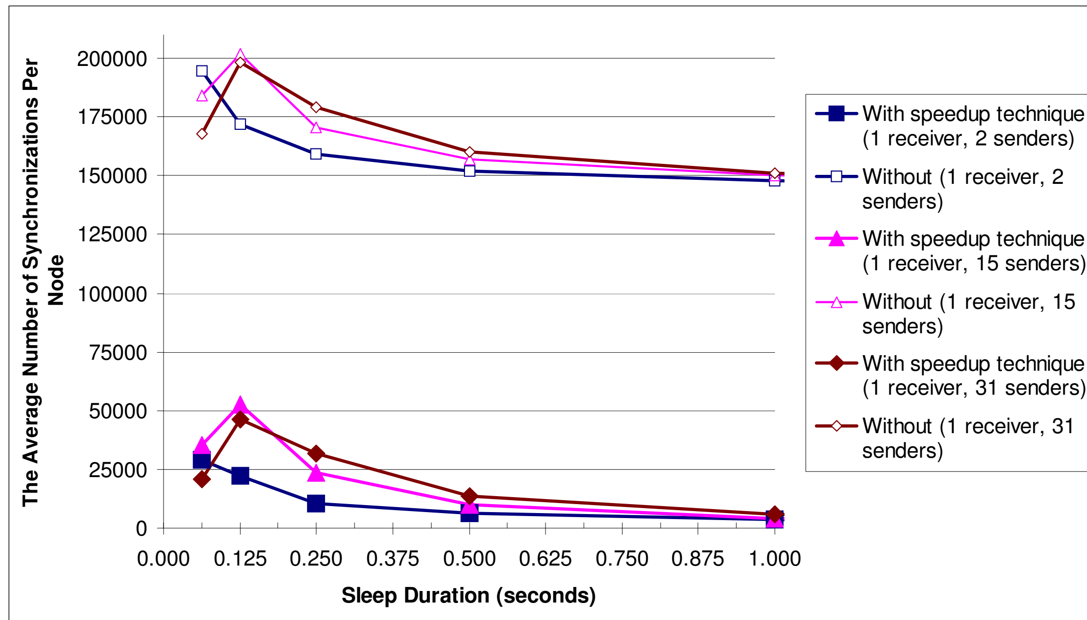

Figure 3.6: Average number of synchronizations per node in one-hop networks during 60 seconds of simulation time (a zoomed in view of Figure 3.2)

average number of synchronization per node with speedup is similar for all network sizes.

3.5.2 Performance in multi-hop networks

In this section, we evaluate the performance of the speedup technique using multi-hop sensor networks. Nodes are laid 20 meters apart on square grids of various sizes. Sender and receivers are positioned on the grids in such a way that nodes of the same types are not adjacent to each other. By setting a maximum transmission range of 20 meters, this setup ensures that only neighboring nodes are within direct communication range of each other. This configuration is very similar to the two dimensional topology in DiSenS [WWM07]. Once again, only senders are duty cycled to keep the experiments simple.

Figure 3.7 shows the average number of synchronizations per node in multi-hop networks during 20 seconds of simulation time. The percentage reductions of the average number of synchronizations per node in multi-hop networks during the 20 seconds of simulation time are shown in Figure 3.8. We can see that there are significant reductions in the average number of synchronizations per node in all the test cases using the speedup technique and the reduction percentages scale with sleep durations.

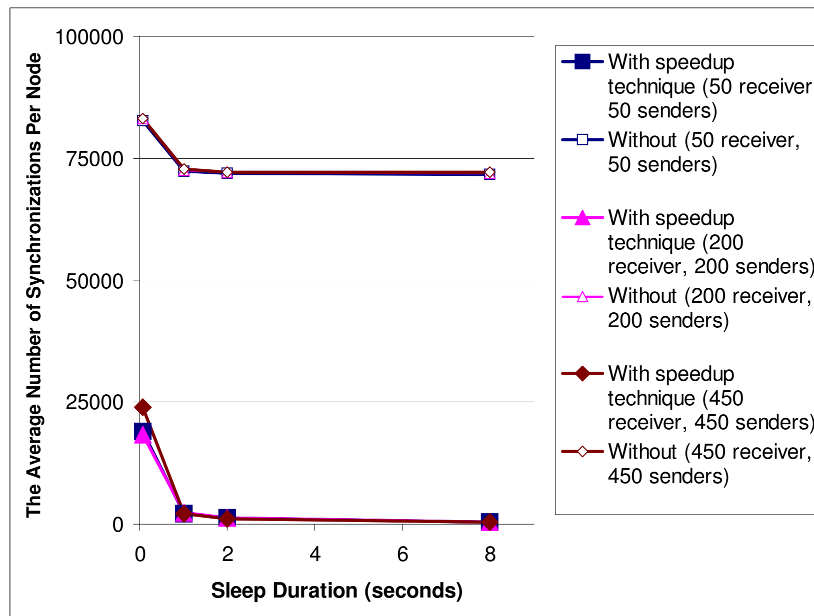

Figure 3.7: Average number of synchronizations per node in multi-hop networks during 20 seconds of simulation time

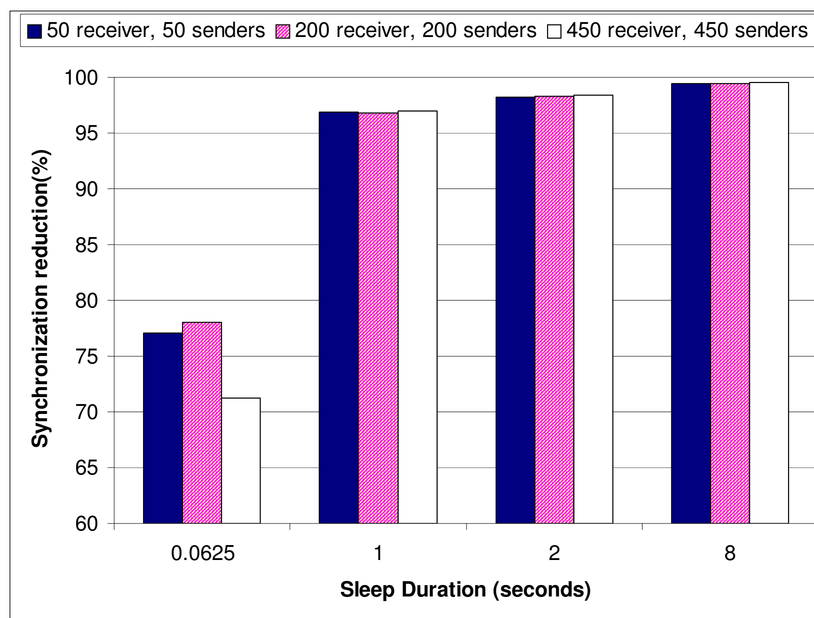

Figure 3.8: Percentage reductions of the average number of synchronizations per node in multi-hop networks during 20 seconds of simulation time

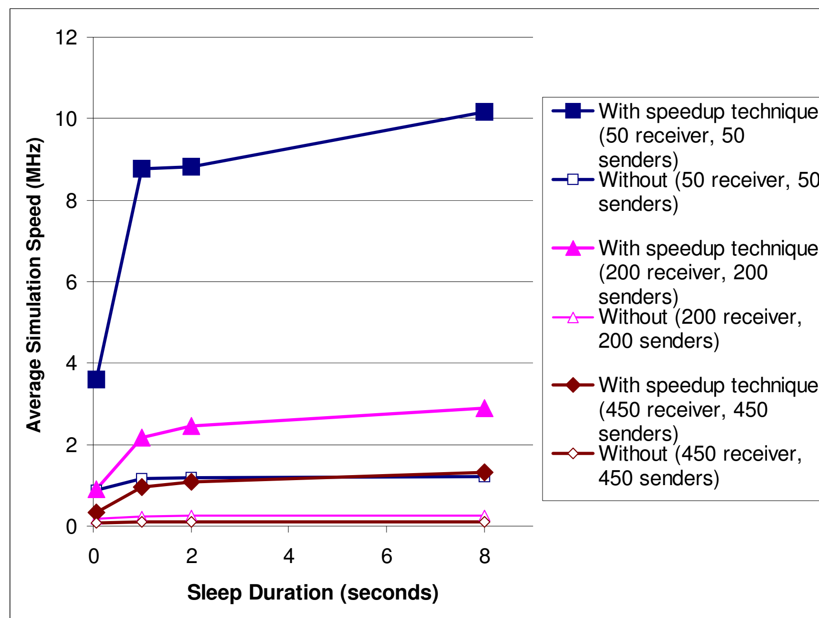

Figure 3.9: Average simulation speed in multi-hop networks

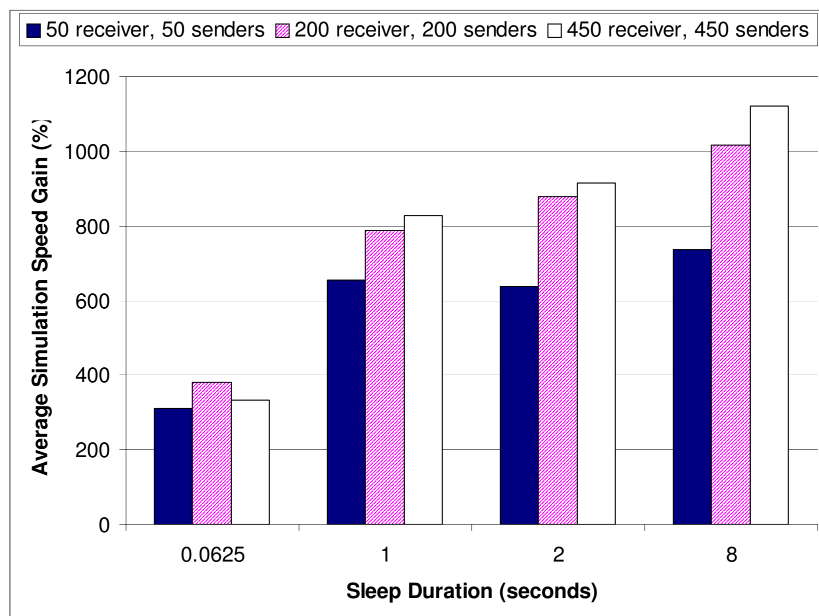

Figure 3.10: Percentage increases of average simulation speed in multi-hop networks

Figure 3.9 and Figure 3.10 indicate that the speedup technique significantly increases average simulation speed in all multi-hop test cases. Compared to the one-hop test results in Figure 3.5, the speed increases scale better with network sizes in multi-hop

tests. This is because our distributed synchronization algorithm has less overhead on sensor networks that have smaller numbers of nodes within direct communication range as described in the end of Section 3.3.

3.6 Related work

Previous work on improving the speed and scalability of WSN simulators can be broadly divided into three categories. The first category focuses on trading simulation fidelity for simulation speed. As described in Chapter 1, it works by giving up a small degree of fidelity for a big reduction of the computational demands of individual simulation models. A good example of this approach is TimeTossim [LAW08]. Our work makes this type of effort scalable on multiple processors/cores.

The second category of work focuses on reducing overheads in parallel and distributed simulations. DiSenS reduces the overheads of synchronizing nodes across computers by using the sensor network topology information to partition nodes into groups that do not communicate frequently and simulating each group on a separate computer [WWM07]. However, this technique only works well if most of the nodes are not within direct communication range as described in the paper. Increasing lookahead time is another commonly used approach to reduce the overheads of the conservative approach [FKM92, Fuj99a, LN02] and increase parallelism. Our technique belongs to this category in the sense that we also improve speed and scalability by increasing the lookahead time. However, our technique is fundamentally different since we use application specific characteristics in a different context to increase lookahead time.

The third category uses special-purpose hardware to improve simulation speed. For example, the IBM Yorktown Simulation Engine can increase the speed of gate-level logic simulations by several orders of magnitude using highly parallel, special-purpose hardware [Den82, KP82, Pfi86]. Our proposed technique is a complement to such approaches.

3.7 Summary

We have described a speedup technique that significantly reduces the number of sensor node synchronizations in distributed simulations of WSNs and consequently improves average simulation speed and scalability of distributed WSN simulators. We

implemented this technique in the PolarLite simulator which is developed on top of the Avrora simulator, a widely used parallel sensor network simulator and conducted extensive experiments. The significant performance improvements on a SMP computer suggest even greater benefits in applying the speedup technique to distributed simulations over a network of computers because of their large overheads in sending synchronization messages across computers during simulations.

Acknowledgements: Chapter 3, in part, has been published as “Improved Distributed Simulation of Sensor Networks Based on Sensor Node Sleep Time” by Zhong-Yi Jin and Rajesh Gupta in DCOSS 08: Proceedings of the 4th ACM/IEEE International Conference on Distributed Computing in Sensor Systems [JG08], pages 204-218. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Exploiting Radio and MAC Level Parallelism in Conservative Simulations

In the previous chapter, we describe how to exploit application level parallelism to improve the performance of conservative WSN simulators. In this chapter, we present two techniques that improve the performance of conservative WSN simulators by exploiting radio and MAC level parallelism. In addition, we describe a probing mechanism that makes it possible to exploit any potential application specific characteristics at the communication layers for synchronization reductions.

4.0.1 Technique to Exploit Radio-level Parallelism

Our radio-level speedup technique exploits the radio off time when a sensor node radio is duty cycled. Radio-level duty cycling works by selectively turning radios on and off. Since radios are one of the most power consuming components of sensor nodes, radio-level duty cycling is ubiquitously used in WSNs to reduce energy consumption and extend working life of energy constrained sensor nodes. Due to its wide applications and the complex tradeoffs in energy savings and communication overheads, radio-level duty cycling is commonly built into energy efficient WSN MACs such as S-MAC [YHE02] and B-MAC [PHC04].

Our radio-level speedup technique is illustrated in Figure 4.1 which shows the progress of simulating two sensor nodes in parallel. In this simulation, Node B turns

its radio off at time T_{S1} and puts it back on at time T_{Sx}. With existing distributed simulators, after running the simulation for T_{W0} seconds of wallclock time, Node A has to wait at T_{S3} for Node B to catch up from T_{S2}, despite the fact that node B will not transmit any packets at T_{S3}. Ideally, we can avoid this unnecessary synchronization by having Node B notify node A at time T_{S1} that its radio is off until T_{Sx}. However, this will not work as it is not possible for Node B to predict the exact radio wakeup time T_{Sx} at T_{S1}. This is because while the radio is off at T_{S1}, the sensor node processor is still running and it can turn the radio back on at any time based on current states, application logics and sensor readings. In other words, it is just not possible for Node B to predict when the radio will be turned on in the future.

Instead of predicting the exact radio wakeup time, our radio-level speedup technique exploits the radio off period by calculating the earliest possible communication time, $T_{EarliestCom}$. $T_{EarliestCom}$ is the earliest time that a turned off radio can be used to send or receive data over wireless channels and can be calculated based on T_{Act}, the amount of time to fully activate a turned off radio. A turned off radio can not be activated instantly for sending or receiving data. It takes time for the radio to be initialized and become fully functional [PSC05]. For example, the CC1000 radio of Mica2 nodes [Cro08] needs 2.45ms to be activated and the CC2420 radio of Telos nodes [PSC05] needs about 1.66ms without counting the SPI acquisition time [Lev06]. The exact delays in terms of numbers of clock cycles are hard coded into WSN MAC protocols and can be easily identified in the source code. For example, in TinyOS 1.1 [HSW+00, TA], B-MAC waits for a total of 34300 clock cycles for the CC1000 radio of MICA2 by calling the `TOSH_uwait` function. While the delays seem to be small, they are significantly larger than typical lookahead times in simulating WSNs. For example, it is about 11 times larger than the 3072 clock cycle lookahead time in simulating Mica2 nodes [TLP05, JG08]. As mentioned, lookahead time is defined as the maximum amount of simulation time that a simulated sensor node can advance freely without synchronizing with other simulated sensor nodes [JG08].

Our radio-level speedup technique works by tracking when sensor node radios are turned on and off. When we detect that a sensor node radio is turned off, we immediately send its $T_{EarliestCom}$ in a clock synchronization message to all neighboring nodes and then repeatedly send the latest $T_{EarliestCom}$ every T_{Act} time until the radio is detected to be turned on. $T_{EarliestCom}$ can be calculated as the sum of current simulation time and T_{Act}.

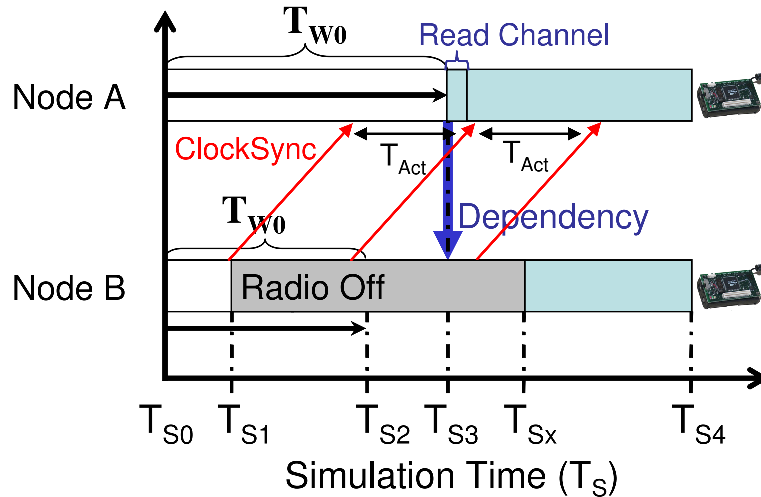

Figure 4.1: The progress of simulating two nodes that are in direct communication range with the radio-level speedup technique

As a result, neighboring nodes no longer need to synchronize with the radio off node until the latest $T_{EarliestCom}$. For example, as shown in Figure 4.1, when we detect that Node B turns its radio off at T_{S1}, we immediately send its $T_{EarliestCom}$ to Node A and repeat that every T_{Act} time which is fixed according to the radio of Node B. The clock synchronization messages are shown as arrows from Node B to Node A in the figure. Upon receiving the second $T_{EarliestCom}$, the lookahead of node A increases to a time beyond T_{S3} and therefore it no longer needs to wait at T_{S3} after T_{W0} seconds of simulation. In other words, Node A knows before T_{W0} that Node B will not be able to transmit any packet at T_{S3}. Since the increase of lookahead time ($T_{Act} - OldLookAheadTime$) may just be a very small fraction of the total radio off period, it is critical to repeatedly send $T_{EarliestCom}$ every T_{Act} time to fully exploit the entire radio off period.

The radio-level speedup technique also reduces the number of clock synchronizations. In distributed simulations, clock synchronization messages are used to send the simulation time of a node to all its neighboring nodes so causality can be maintained and suspended waiting nodes can be revived. To maximize parallelism in simulations, a node needs to send 1 clock synchronization message for every lookahead time of its neighboring nodes [TLP05, WWM07, JG08]. Since our radio-level speedup technique increases the lookahead times of neighboring nodes, the number of clock synchronizations can be greatly reduced. For example, in the case of simulating Mica2 nodes, one $T_{EarliestCom}$ message can increase lookahead time by a factor of 11 and therefore eliminates 10 clock

synchronization messages.

4.0.2 Technique to Exploit MAC-level Parallelism

While the radio-level speedup technique takes advantage of physical delays in WSN radios, our MAC-level speedup technique exploits the random backoff behaviors of WSN MACs. Almost all WSN MACs need to perform random backoffs to avoid concurrent transmissions [YHE02, PHC04]. For example, before transmitting a packet, B-MAC would first perform an initial backoff. If the channel is not clear after the initial backoff, B-MAC needs to repeatedly perform congestion backoffs until the channel is clear. Because a MAC will not transmit any data during backoff periods, we are able to exploit the backoff times for speedups. Although the backoff times are random and MAC specific, they are usually a lot longer than typical lookahead times in simulating WSNs. For example, in the case of B-MAC, the default initial backoff is 1 to 32 times longer than the 3072 clock cycle lookahead time in simulating Mica2 nodes. The default congestion backoff is 1 to 16 times longer in B-MAC.

Our MAC-level speedup technique is illustrated in Figure 4.2 which is similar to Figure 4.1 except Node B enters into a backoff period from T_{S1} to T_{S4}. The MAC-level speedup technique works by detecting the start and the duration of a backoff period. When the start of a backoff period is identified, the end time of the backoff period is first calculated based on the duration of the period and then sent to the neighboring nodes. This effectively increases the lookahead times of neighboring nodes and helps to eliminate unnecessary synchronizations. For example, in order to avoid the unnecessary synchronization of Node A at T_{S3} after running the simulation for T_{W0} seconds of wall-clock time, our MAC-level speedup technique first detects at T_{S1} the start of Node B's backoff period as well as the duration of the backoff period. Then we compute the end time of the backoff period and send that in a clock synchronization message to Node A. Once Node A knows that Node B will not transmit until T_{S4}, it no longer needs to wait at T_{S3}. Similar to the radio-level speedup technique, the MAC-level technique also reduces the number of clock synchronizations, which provides additional speedup. We discuss how to detect the start and the duration of a random backoff period in Section 4.1.

The MAC-level speedup technique is a good complement to the radio-level technique as WSNs usually have very bursty traffic loads. Nodes in a WSN usually do not communicate frequently and can duty cycle their radios extensively until certain trigger-

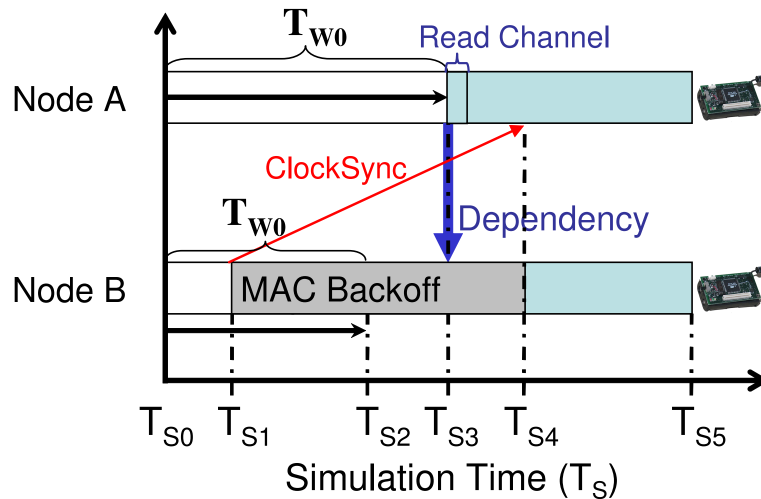

Figure 4.2: The progress of simulating two nodes that are in direct communication range with the MAC-level speedup technique

ing events occur. Once triggered by those events, nodes need to actively communicate and interact with each other to accomplish certain tasks. Our MAC-level technique is most effective when wireless channels are busy.

4.1 Implementation

The proposed speedup techniques are implemented in PolarLite, a distributed simulation framework that we developed based on Avrora as described in Chapter 3. PolarLite provides the same level of cycle accurate simulation as Avrora but uses a distributed synchronization engine instead of Avrora's centralized one. This makes it possible to implement and evaluate our speedup techniques in a distributed simulation environment. The synchronization engine of PolarLite is based on the distributed synchronization algorithm described in Chapter 3. With this algorithm, nodes can be synchronized separately according to their own lookahead times. In other words, if a node is not accessing the wireless channel, it only needs to communicate to neighboring nodes its simulation time and does not need to wait for any other nodes. As with Avrora, PolarLite allocates one thread for each simulated node and relies on the Java virtual machine (JVM) to assign runnable threads to any available processors on an SMP computer.

To implement the radio-level speedup technique, we need to detect when radios

are turned on and off. In discrete event driven simulations, the changes of radio states are triggered by events and can be tracked. For example, in our framework, we detect the radio on/off time by tracking the IO events that access the registers of simulated radios.

Detecting MAC backoff times and durations for the MAC-level speedup technique are considerately more difficult. The backoffs are MAC and application specific and generally do not correlate to any unique events or actions that can be easily tracked. In addition, the backoff durations are completely random. One possible solution to this problem is to insert special code into the source code of an application that is to be simulated. These special pieces of code are compiled into the application and used to report to the simulator the MAC backoff times and durations during simulations. However, this technique does not work for cycle accurate simulators like ours.

Cycle accurate sensor network simulators [PBM+04, TLP05, WWM07, JG08] offer the highest level of fidelity among all types of sensor network simulators. They provide simulation models that emulate the functions of major hardware components of a sensor node, mainly the processor. Therefore, they take as inputs the same binary code (images) that are executed by real sensor nodes. To detect backoff times and durations without changing the source code of the applications under simulation, we develop a generic probing mechanism based on pattern matching to expose the internal states of sensor network applications during simulations.

Our probing mechanism works by first using patterns to pinpoint from compiled applications the machine instructions that represent events of interest during the start of a simulation. The identified instructions are then replaced by corresponding "hook" instructions to report the internal states of the applications, such as the backoff durations, to a simulator during simulations. Hook instructions are artificial instructions that behave exactly the same as the original instructions they replace except they will pass to a simulator the memory locations or registers accessed by the original instructions during simulations. The values stored in those locations are the internal states of the applications that correspond to the events of interest. Since an instruction may access multiple locations, we associate with each pattern a list that indicates the operands of interest based on their order in the instruction. Therefore, an instruction may be translated into different hook instructions according to the list. To maintain cycle accuracy, our simulator ensures that the hook instructions consume the same number of clock

cycles as the original instructions.

Our current implementation of the probing mechanism is largely based on existing constructs from Avrora. In Avrora, a compiled program (object file) is disassembled first before simulation and each disassembled instruction is loaded into a separate instruction object (Java object). Once an instruction of interest is identified with pattern matching, we encapsulate the corresponding instruction object in a new hook instruction object and attach to the hook instruction object a probe object created specifically for the pattern. When executed, the hook instruction invokes the original instruction first and then calls the probe attached to it. It is the specific probe that turns a regular hook instruction into a unique hook instruction and reports values of interest to a simulator during simulations.

We do not use addresses to identify instructions of interest because addresses tend to vary across compilations after source code changes, even if the changes are at places not related to the instructions. With pattern matching, we only need to create a set of patterns once if the corresponding source code does not change. For example, if an application is written with TinyOS, the instructions that assign backoff durations to B-MAC are part of the OS, regardless of whether the backoffs are calculated by default functions in the OS or user supplied functions in the application. Therefore, we only need to create a set of patterns once for each version of TinyOS to track the backoff times in B-MAC during simulations.

We use regular expressions for pattern matching. To uniquely identify an instruction, we need to match additional instructions before or after that instruction as well. In the current implementation, the backoff matching process for B-MAC is hard coded in our simulator. To match the initial backoff, we first locate the block of code that corresponds to the *send* function in a disassembled program by using the function name (symbol name). The *send* function is a part of TinyOS and is where the initial backoff calculation function is invoked. Then we match within this code block a continuous sequence of instructions (*sts,sts,sts,lds,out,and,brne,rjmp*) which are instructions that immediately follow the initial backoff calculation code. Note that we only need to match the names of the instructions in the sequence. Once this pattern is found, the value for initial backoff can be tracked via the first 2 *sts* instructions in the matched code. Similarly, we can identify the instructions that store congestion backoffs. For simplicity, we consider that MAC backoffs start at the times that the hook instructions report the

Table 4.1: Radio off periods under different duty cycling modes of B-MAC

Duty cycling Mode	Radio off Time (ms)
0	0
1	20
2	85
3	135
4	185

backoff durations. It is safe to do so as no data will be sent from this point on until the end of the backoff periods.

Note that we can not simply use the symbol names of the backoff calculation functions for pattern matching because these functions are in-lined by the compiler. However, there are always some caller functions in TinyOS, such as the *send*, that are not in-lined due to space and other constrains. Based on these functions, we can create patterns that remain the same as long as the functions do not change.

4.2 Evaluation

We conduct a series of experiments to evaluate the performance of our speedup techniques. All experiments are carried out on an SMP server running Linux 2.6.24. The server features a total of 8 cores on 2 Intel Xeon 3.0GHz CPUs and 16GBytes of RAM. Sun's Java 1.6.0 is used to run all experiments. Simulation speed is calculated using Equation (4.1). Note that the numerator of Equation (4.1) is the total simulation time in units of clock cycles and consequently the calculated speed is in units of Hz.

$$Speed = \frac{total\ number\ of\ simulated\ clock\ cycles}{(execution\ time) \times (number\ of\ nodes)} \tag{4.1}$$

All of the experiments are based on the CountSend (sender) and CountReceive (receiver) programs from the TinyOS 1.1 distribution. They are similar in nature to the programs used by other WSN simulators in evaluating their performance [LLWC03, TLP05, WWM07]. CountSend broadcasts at a fixed interval the value of a continuously increasing counter. CountReceive simply listens for messages sent by CountSend and displays the last received value on LEDs. The programs are executed on simulated Mica2 nodes [Cro08] and the starting time of each node is randomly selected between 0 and 1 second of simulation time to avoid any artificial time locks. All simulations are

run for 300 seconds of simulation time and for each experiment we take average of three runs as the results.

4.2.1 Performance of Radio-level Technique

For experiments in this section, we modify the sender and receiver programs slightly to enable B-MAC's built-in radio-level duty cycling feature. This can be done by calling the SetListeningMode and SetTransmitMode functions of TinyOS 1.1 at start. B-MAC supports a total of 7 radio-level duty cycling modes in TinyOS 1.1 and the 5 modes used in our experiments are shown in Table 4.1. Once enabled, B-MAC turns a radio off periodically for a duration corresponding to the duty cycling mode. The radio is turned back on either when there are data to transmit or a radio off period ends. The radio is turned off again once there are no pending packets to transmit and the channel is clear for a fixed period of time. In the case of TinyOS 1.1 and Mica2 nodes, the channel clear time is the amount of time to transmit 8 bytes over the radio [HSW+00, TA].

Speed and Scalability with respect to the number of processors

Our first set of experiments evaluates the performance of the radio-level speedup technique over different numbers of processors, or cores in this case. For these experiments, we simulate a WSN of 32 nodes that are within direct communication range using 2 to 8 processors. The 32 nodes are set up in such a way that one node is configured as a sender and the rest as receivers. Since all nodes are within direct communication range, any one of the nodes can be chosen as the sender. The frequency that the sender transmits packets is varied for different experiments. The radio-level duty cycling modes of all nodes are set to 3. For comparisons, we conduct the same experiments using Avrora, PolarLite without any speedups and PolarLite with the radio-level speedup.

As a baseline, Figure 4.3 compares the speeds of simulating with Avrora and PolarLite running the radio-level speedup technique. We can see that PolarLite running the radio-level speedup technique is considerably faster than Avrora (up to 544% or 6.44 times) and scales with the number of processors. In contrast, the speeds of simulating with Avrora decrease with increasing number of processors in this set of experiments due to larger synchronization overheads[1].

[1]Our results are different from results of similar experiments in [TLP05] as our experiments use faster 3.0GHz CPUs, compared to 900MHz ones of theirs.

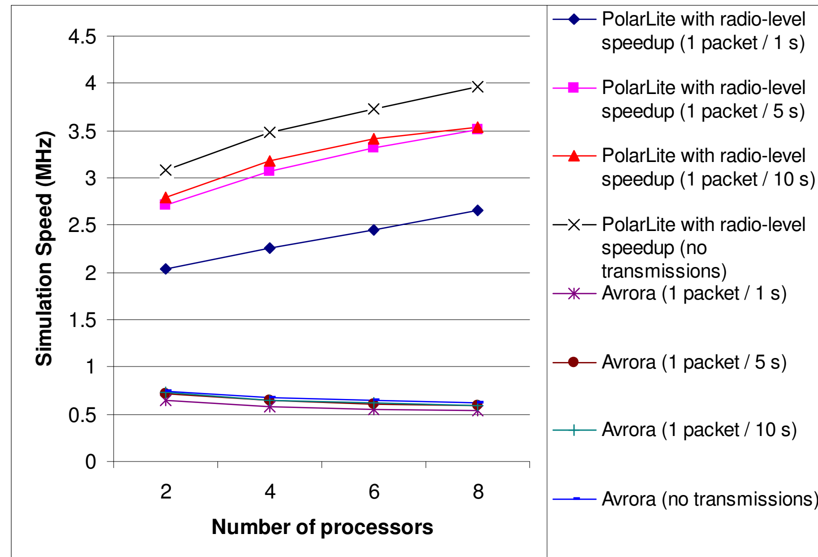

Figure 4.3: Speed of simulating with Avrora and PolarLite running the radio-level speedup (1 sender 31 receivers, mode 3)

We can also see in Figure 4.3 that the speeds of simulating with our radio-level speedup technique increase with transmission intervals. This is because our radio-level speedup technique is based on exploiting radio off time and large transmission intervals increase that. Note that at a given radio-level duty cycling mode, increasing the transmission intervals will also increase the radio off time of the receivers because radios have to be left on when receiving packets. However, as shown in Figure 4.3, the percentage increases of simulation speed with the radio-level speedup technique decrease quickly with increasing transmission intervals. This is due to the fact that when transmission intervals increase, the radio off time is determined more by the radio-level duty cycling mode than by the transmission intervals.

Figure 4.4 shows the percentage reductions of synchronizations based on numbers collected by running with and without the radio-level speedup technique in PolarLite. For accurate evaluations, we only show synchronization reductions within our PolarLite framework because PolarLite and Avrora are based on different synchronization algorithms and our speedup techniques are only implemented in PolarLite.

As shown in Figure 4.4, the percentage reductions of synchronizations are significant in all cases and actually grow very slowly with the number of processors. This is because more nodes can be simulated in parallel when the number of processors in-

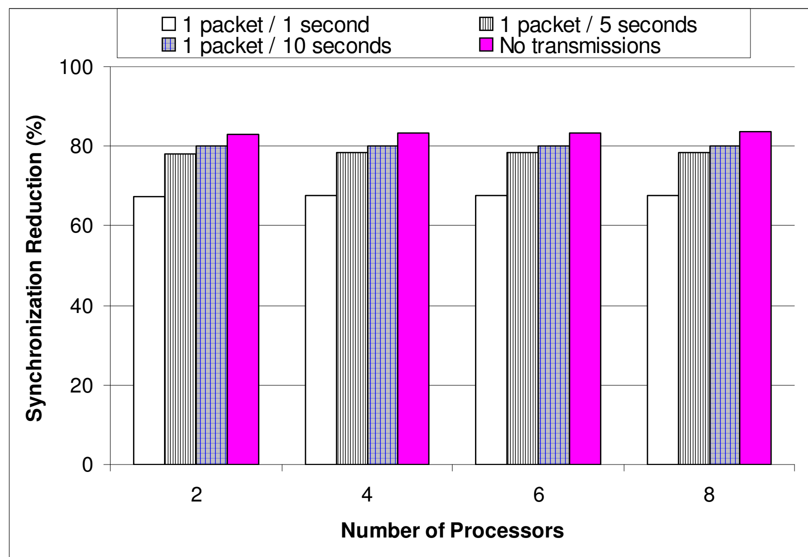

Figure 4.4: Percentage reductions of synchronizations using the radio-level speedup technique in PolarLite (1 sender 31 receivers, mode 3)

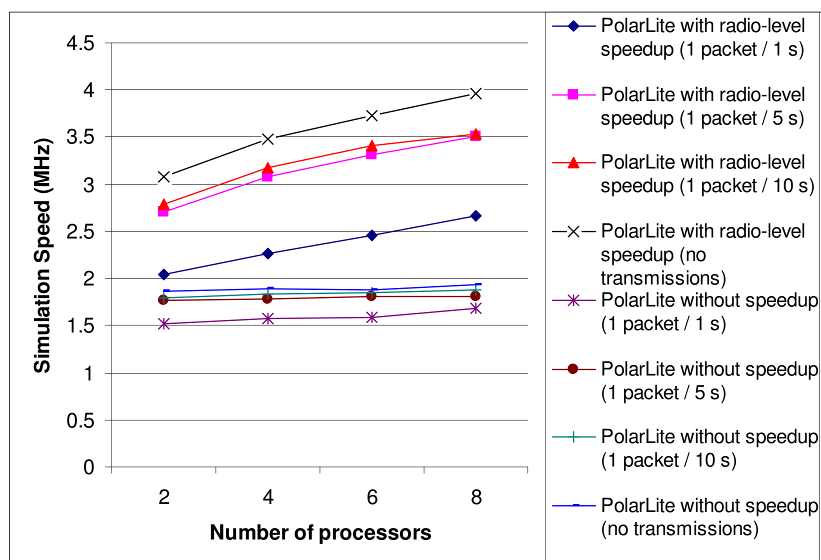

Figure 4.5: Speed of simulating with and without the radio-level speedup technique in PolarLite (1 sender 31 receivers, mode 3)

creases. As a result, our radio-level speedup technique has more radio sleep time to exploit at a given time. Although the reduction numbers are very close with respect to the number of processors, simulation speeds increase significantly with the number of processors in Figure 4.5 which shows the speed of simulating with and without the radio-level speedup technique in PolarLite using different number of processors. The reason is that per-synchronization overheads increase with the number of processors due to high inter-processor communication overheads.

As shown in Figure 4.5, using the radio-level speedup technique increases simulation speeds significantly (up to 111%) in PolarLite. Comparing with Figure 4.3, we observe that PolarLite alone without any speedup techniques is faster than Avrora in these experiments. This is because our distributed synchronization algorithm (Section 4.1) can provide more parallelism by allowing nodes to be synchronized separately according to their own lookahead times. Avrora on the other hand synchronizes all nodes together at a fixed time interval. However, even using the distributed synchronization algorithm, PolarLite alone does not scale well with the number of processors as shown in Figure 4.5. Using the radio-level speedup technique significantly improves scalability.

Speed and Scalability with respect to network sizes and radio off times

We also evaluate the radio-level speedup technique over WSNs of different sizes and radio sleep durations (radio-level duty cycling modes). Similar to the setups in Section 4.2.1, nodes in these experiments are within direct communication range and only one node is configured as the sender. The sender transmits a packet every 10 seconds to the rest of receiver nodes. Figures 4.6, 4.7 and 4.8 show the results of simulating with or without the radio-level speedup technique in PolarLite using all 8 processors.

Figure 4.6 shows significant percentage reductions of synchronizations using the radio-level speedup technique. There are no reductions when the radio is constantly on because the radio-level speedup technique works by exploiting radio off time. Figure 4.6 also shows that the reduction percentages scale with radio off durations since larger durations bring more radio off time. While the reduction percentages are about the same for all network sizes at a given radio off duration, the percentage increases of simulation speed actually grow with network sizes according to Figure 4.6. This is because in a network where all nodes are in direct communication range, the total number of synchronizations in a distributed simulation is in the order of $N * (N-1)$ where N

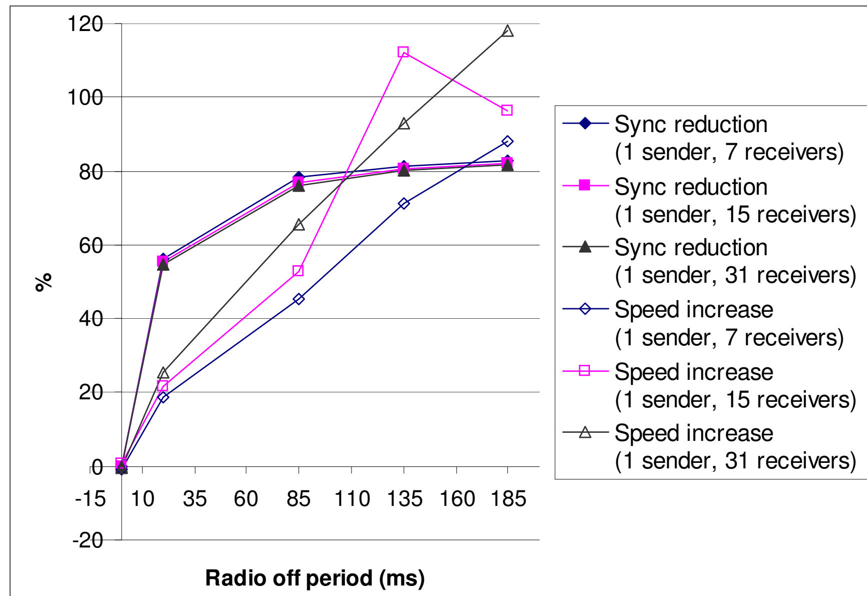

Figure 4.6: Percentage reductions of synchronizations and percentage increases of simulation speed using the radio-level speedup technique in simulating WSNs of different sizes and radio off times in PolarLite (1 packet/10 seconds, 8 processors)

is the network size. Therefore, the total number of reductions in the experiments grows with network sizes. This can be seen clearly in Figure 4.7 which shows the total number of synchronizations in logarithmic scale.

We can also see from Figure 4.6 that the percentage increases of simulation speed scale well with radio off durations. The case of simulating 16 nodes with 135ms radio off duration appears to be an outlier, showing a much higher increase in simulation speed than normal. The figure also shows that in this case, the reduction in synchronizations is not unusual to cause a higher simulation speedup. We find that this outlier point is actually caused by a slow simulation speed in PolarLite without using the radio-level speedup technique. In fact, when increasing the sleep duration from 85ms to 135ms, the simulation speed actually decreases from 37.45MHz to 37.10MHz. So, when we apply radio-level speedup to this case, the relative increase becomes larger than ordinary. This shows that the proposed radio-level speedup is effective even when the baseline simulation in PolarLite can not benefit from the increased radio off time.

Finally, we evaluate the scalability of the radio-level speedup technique over larger WSNs under a transmission rate of 1 packet/10 seconds and a radio-level duty cycling

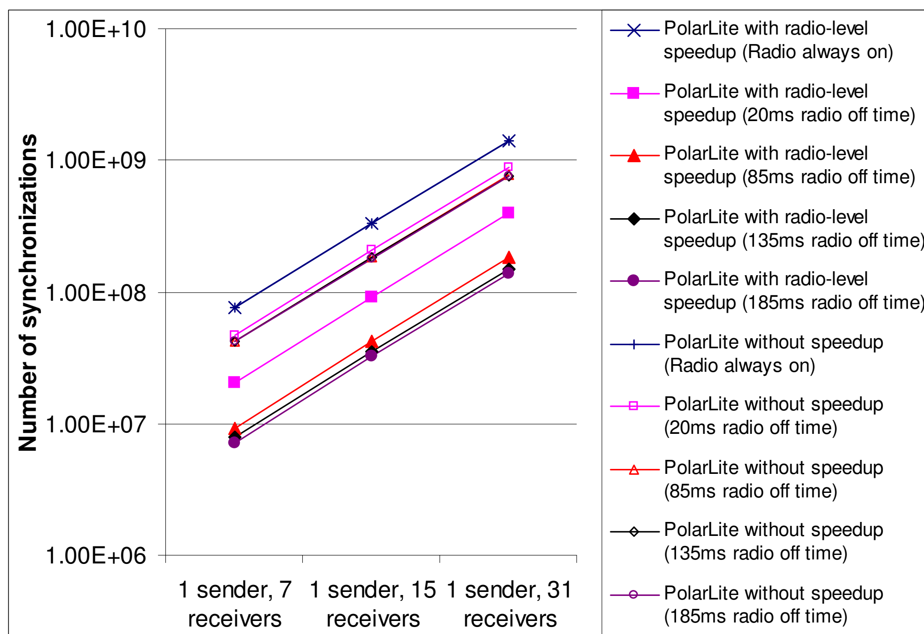

Figure 4.7: Total number of synchronizations in simulating WSNs of different sizes and radio off times with and without the radio-level speedup technique in PolarLite (1 packet/10 seconds, 8 processors)

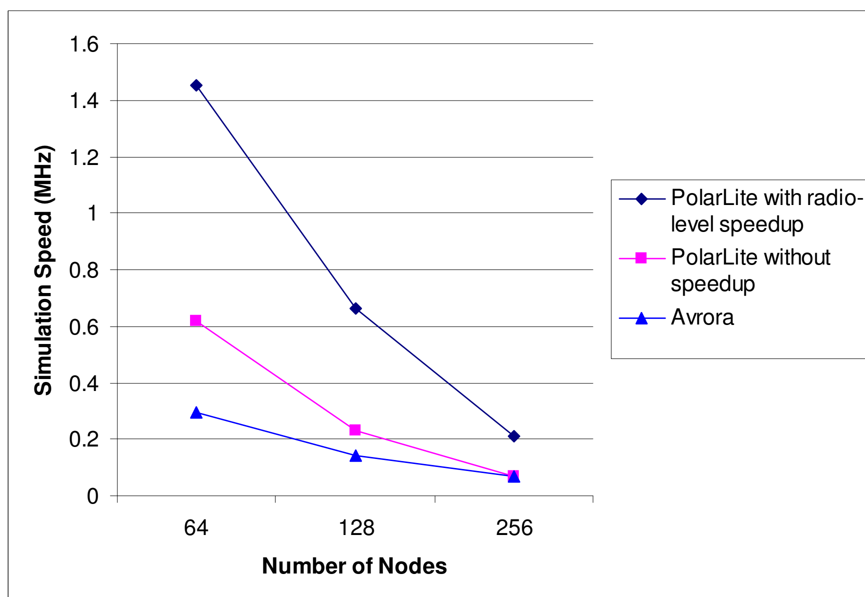

Figure 4.8: Speed of simulating large WSNs (1 packet/10 seconds, 8 processors, mode 3)

mode of 3, using all 8 processors. The results are shown in Figure 4.8. We can see that the radio-level speedup technique increases simulation speed in large WSNs as well. It provides a 197% increase of simulation speed when simulating 256 nodes in PolarLite. That is an additional 106% improvement over the 91% speed increase in simulating 32 nodes under the same setup in Figure 4.6. In other words, although simulation speeds decrease with network sizes due to the limited computational power of our server, the percentage increases of simulation speed using the radio-level speedup technique still grow with network sizes.

4.2.2 Performance of MAC-level Technique

The performance of our MAC-level speedup technique depends on how busy wireless channels are and how often sensor nodes transmit around the same time. Instead of evaluating with a large number of scenarios, we study the maximum speedup that can be achieved in simulating a WSN with the MAC-level speedup technique. For experiments in this section, we enable CountSend to send as fast as possible by modifying CountSend such that it sends out a new packet as soon as it is notified by the MAC that the previous packet is sent. We also disable the radio-level duty cycling for both CountSend and CountReceive.

We simulate two WSNs that have 1 receiver and 31 or 63 senders using Avrora, PolarLite without speedups and PolarLite with the MAC-level speedup. Unless explicitly specified, the default backoff calculation functions in TinyOS 1.1 are used for the senders. The results are shown in Figures 4.9, 4.10 and 4.11.

Speed and Scalability with respect to the number of processors and backoff times

We can see from Figure 4.9 that the MAC-level speedup technique reduces synchronizations by about 44% to 47% in PolarLite. As a result, it brings a speedup of 14% to 31% (96% to 323% compared to Avrora) using the default backoff calculation functions of TinyOS 1.1 as shown in Figure 4.10. However, the default backoff windows are not large enough for our experiments since we observe a significant amount of colliding transmissions causing dropped packets. This limits the performance of our MAC-level speedup technique as nodes may transmit at the same time without backoffs. To further investigate this, we perform the same experiments by doubling the sizes of the default

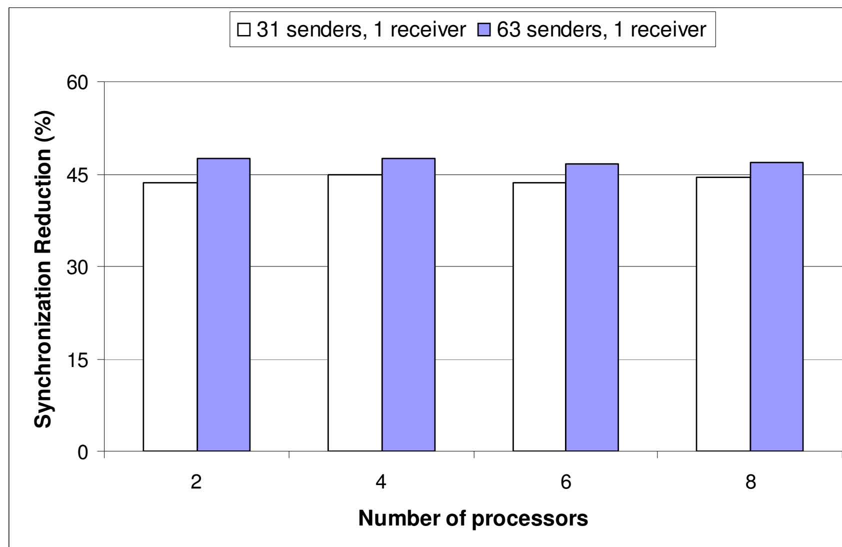

Figure 4.9: Percentage reductions of synchronizations with MAC-level speedup on WSNs of different sizes in PolarLite (No duty cycling)

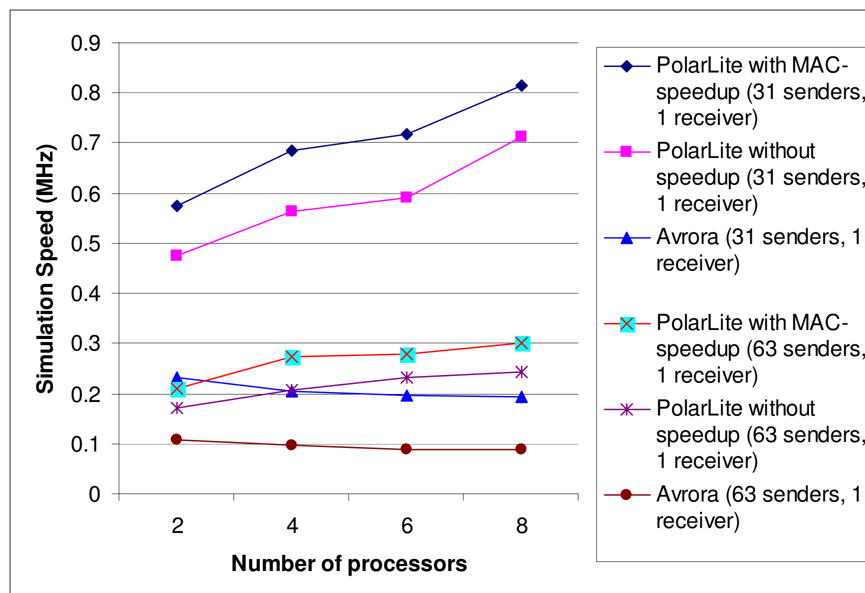

Figure 4.10: Speed of simulating 2 WSNs with Avrora, PolarLite and PolarLite + MAC-speedup (No duty cycling)

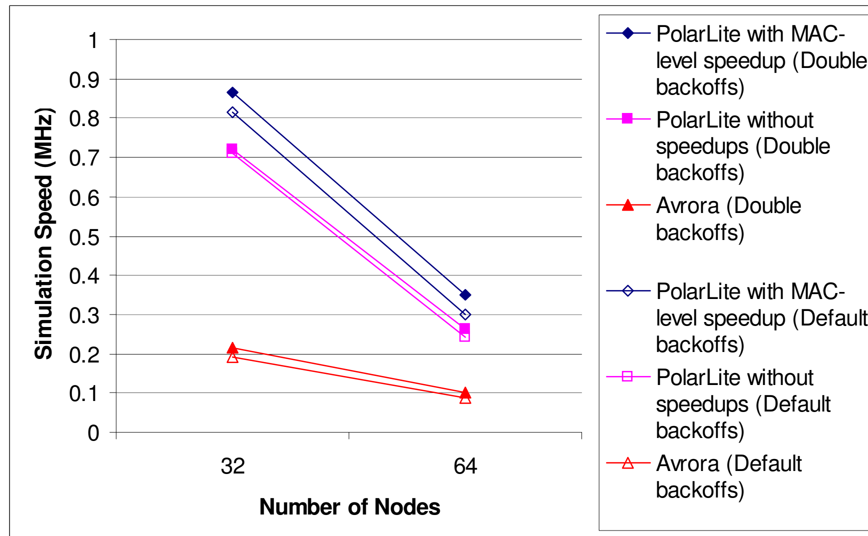

Figure 4.11: Speed of simulating with MAC-level speedup on WSNs using default and double sized backoff windows (No duty cycling, 8 processors)

backoff windows and the results are shown in Figure 4.11. We can see that our MAC-level speedup technique brings more significant increases of simulation speed with larger backoff windows. We can also see that as the number of nodes increases, the speeds of simulating with the MAC-level speedup technique drop faster than with Avrora. This is because given the small backoff window sizes, the number of colliding transmissions increases quickly with the network size in these setups where nodes transmit as fast as possible.

Speed and Scalability with respect to network sizes

Figure 4.9 also shows that the percentage reductions of synchronizations using the MAC-level speedup technique increase with network sizes in PolarLite. As explained in Section 4.2.1, the total number of synchronizations in these experiments is in the order of the square of the network size. Therefore, the total number of reductions is very significant when the network size doubles from 32 to 64. We can see in Figure 4.10 that the percentage increases of simulation speed using the MAC-level speedup technique scale with network sizes even with the default backoff windows. We notice the unusually low increase of speed in simulating with 6 processors. Since the percentage reductions of synchronizations are consistent according to Figure 4.9, we believe this is caused by

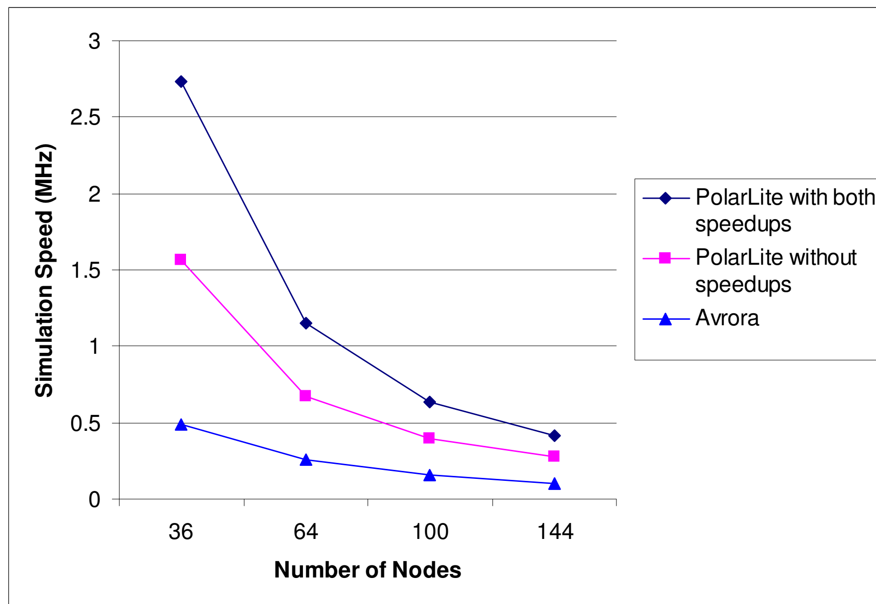

Figure 4.12: Speed of simulating with Avrora, PolarLite without speedups and PolarLite with both speedup techniques (8 processors)

the asymmetrical use of all 4 cores of 1 CPU and 2 cores of another CPU in our server.

4.2.3 Performance with Both Techniques

We evaluate the combined performance of our speedup techniques with a real world scenario. In this scenario, we simulate a WSN service that floods data to every node in a WSN. This service works by having every node in a WSN relay, by broadcasting, messages it receives. To avoid sending duplicate messages, a node only relays messages with IDs greater than the largest IDs of the messages it has already sent. For experiments in this section, we modify CountReceive to relay messages the way we just described.

In our experiments, we simulate WSNs that have nodes laid 3 meters apart on square grids of different sizes. For each of the WSNs, a corner node is configured as the sender and the rest of nodes are configured as relaying nodes running the modified CountReceive program. The sender transmits a new packet every 20 seconds with an increasing ID. The radio-level duty cycling modes of all nodes are set to 4 (Table 4.1) and the backoff windows are doubled from TinyOS 1.1 defaults. The transmit range of all nodes is set to 19 meters. We conduct the experiments with all 8 processors and the

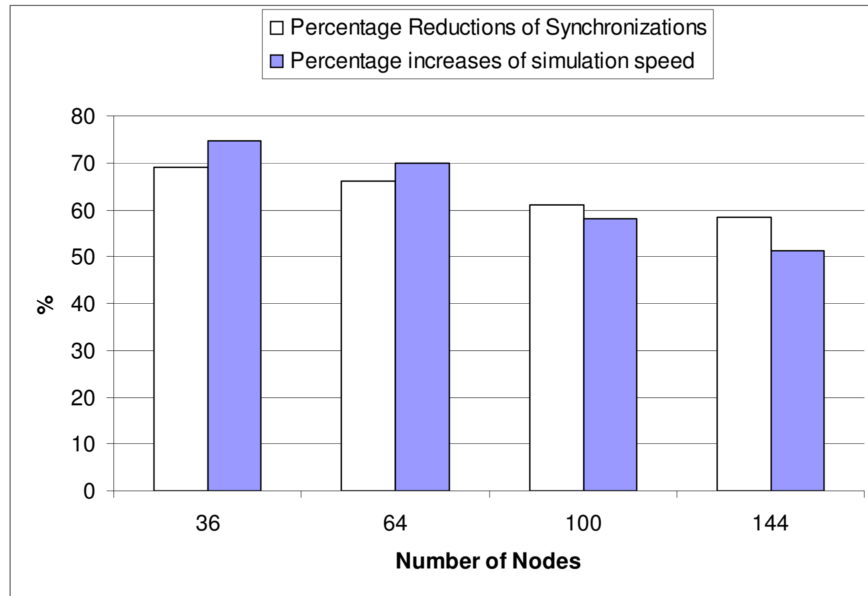

Figure 4.13: Percentage increases of simulation speed and percentage reductions of synchronizations with both speedup techniques in PolarLite (8 processors)

results are shown in Figures 4.12 and 4.13.

As shown in Figures 4.12 and 4.13, PolarLite running both speedup techniques is significantly faster and provides a speedup of 51% to 75% over PolarLite alone and 289% to 462% compared to Avrora. We can also see from Figure 4.13 that the speedup techniques reduce synchronizations significantly by 58% to 70%. However, we observe that the reduction percentages decrease with increasing network sizes. This is caused by our simple flooding protocol. As the network size increases, the number of relaying messages grows as well. Although a node does not transmit the same message twice, it can be forced to receive the same message multiple times from different neighboring nodes. In other words, a transmitting node can keep all nodes within its communication range from turning off their radios. This significantly reduces the sleep time of the nodes and lowers the performance of our radio-level speedup technique. However, even under this setup, our speedup techniques still provide significant increases of simulation speed. This is because our MAC-level speedup technique benefits from an increasing number of backoffs in the larger networks. In practice, more advanced protocols are usually used to reduce the number of unnecessary relaying messages. Therefore, we expect significant better performance with the speedup techniques in those cases.

4.3 Related work

In Chapter 3, we describe a technique that uses sensor node sleep time to reduce the number of synchronizations. As demonstrated, using the node-sleep-time-based technique can significantly increase speed and scalability of distributed WSN simulators. However, the technique is only able to exploit for speedup the time when both the processor and the radio of a sensor node are off. This is because it is not possible to predict the exact radio wakeup time when the processor is running. As a result, we can not apply the node-sleep-time-based technique if the radio is on or if the sensor node processor is kept alive by tasks such as reading and monitoring sensor inputs.

The techniques we propose in this chapter are orthogonal to the node-sleep-time-based technique. They are not bounded by the number of neighboring nodes and are effective even when the processors or the radios of nodes are active. While the techniques are developed specifically for simulating WSNs, they can also be applied to any general distributed network simulators such as *ns-3* [Hen08] for improved performance in simulating wireless networks.

4.4 Summary

We have described two speedup techniques that significantly improve the speed and scalability of distributed sensor network simulators by reducing the number of sensor node synchronizations during simulations. We implemented the techniques in PolarLite, a cycle accurate distributed simulation framework based on Avrora. The significant improvements of simulation performance on a multi-processor computer in our experiments suggest even greater benefits in applying our techniques to distributed simulations over a network of computers because of their large overheads in sending synchronization messages across computers during simulations.

We have also developed a general probing mechanism that can expose the internal states of any sensor network applications during simulations. By knowing the internal states during simulations, we can exploit any application specific characteristics at the communication layers for the increase of lookahead time and as a result, improve simulation speed and scalability.

Acknowledgements: Chapter 4, in part, has been published as “Improving the speed

and scalability of distributed simulations of sensor networks" by Zhong-Yi Jin and Rajesh Gupta in IPSN 09: The 8th ACM/IEEE International Conference on Information Processing in Sensor Networks [JG09a], pages 169-180. The dissertation author was the primary investigator and author of this paper.

Chapter 5

A New Synchronization Scheme for Conservative Simulations

In Chapters 3 and 4, we exploit parallelism at the application, radio and MAC levels to increase the performance of conservative WSN simulators. In this chapter, we take a new approach and present LazySync, a conservative synchronization scheme that further reduces the overheads of the conservative approach in simulating WSNs by identifying and eliminating unnecessary synchronizations during simulations.

5.1 Lazy Synchronization Scheme

In distributed simulations, each sensor node is commonly simulated in a separate thread or process. To maximize parallelism, a running node should try to prevent other nodes from waiting by communicating its simulation progress to those nodes as early as possible (AEAP). If a node has to wait for other nodes due to variations in simulation speeds, the thread/process simulating the waiting node should be suspended so the released physical resources can be used to simulate some other non-waiting nodes[1]. For maximum parallelism, suspended nodes need to be revived AEAP once the conditions that the nodes wait for are met. For example, Node A in Figure 1.1 should synchronize with Node B immediately after it advances past T_{S1} to resume the simulation of Node B.

[1]For synchronization purposes, a non-waiting node refers to a node that is not waiting for any synchronization events. It may still be ready, active or inactive in a given simulation.

The AEAP synchronization scheme is adopted by most existing distributed WSN simulators [TLP05, WWM07, JG08]. It is commonly implemented [TLP05, JG08] by periodically sending the simulation time of every non-waiting node to all its neighboring nodes, which are nodes that are within its direct communication range. Ideally, the clock synchronization period should be as short as possible for maximum parallelism. However, due to the overheads in performing clock synchronizations [JG08], the synchronization period is commonly set to be the minimal lookahead time, which is the smallest possible lookahead time in the simulation. As mentioned, lookahead time is the maximum amount of simulation time that a simulated sensor node can advance freely without synchronizing with any other simulated sensor nodes [Fuj99a]. For example, in the case of simulating Mica2 nodes [Cro08], the minimal lookahead time is the lookahead time of nodes with radios in the listening mode. It is equal to the amount of time to receive one byte over Mica2's CC1000 radio [JG08] and is equivalent to 3072 clock cycles of the 7.32728MHz AVR microcontroller in Mica2. Therefore, when simulating a network of Mica2 nodes, every non-waiting node needs to send its simulation time to all its neighboring nodes every 3072 clock cycles. Depending on simulator implementations, once the simulation time is received by a neighboring node, some mechanisms will be triggered to save the received time and compute the earliest input time (EIT) [CM81]. EIT represents the safe simulation time that the neighboring node can be simulated to. If the neighboring node happens to be waiting, then it will also be revived if its EIT is not less than the wait time. To be revived AEAP, a waiting node commonly sends its waiting time to the nodes that it depends on before entering into the suspended state. By doing so, the depending nodes can send their simulation time to the waiting node immediately after they advance past the waiting time.

5.1.1 Limitations of AEAP Synchronization Scheme

While the AEAP synchronization scheme is sound in principle, its effectiveness is based on the assumption that there is always a free processor available to simulate every revived node. However, this is generally not the case in practice as the number of nodes under a simulation is usually a lot larger than the number of processors used to run the simulation. As a result, the AEAP synchronization scheme may slow simulations down in many simulation scenarios by introducing unnecessary clock synchronizations. For example, Figure 5.1 shows the progress of simulating in parallel 3 nodes that are

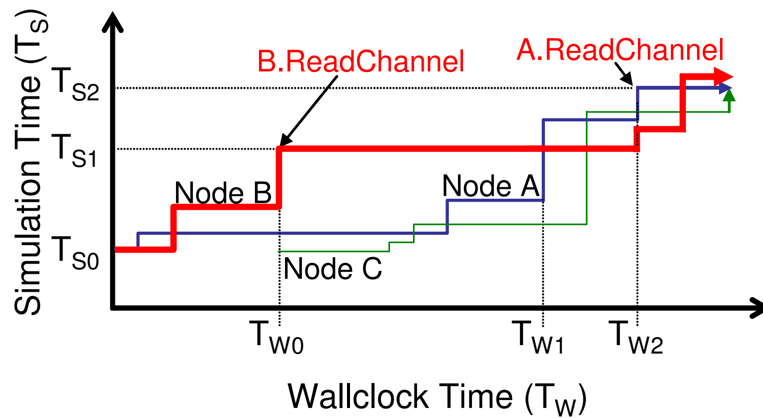

Figure 5.1: The progress of simulating in parallel a wireless sensor network with three nodes that are in direct communication range of each other on 2 processors.

in direct communication range of each other on 2 processors. In the simulation, Node A and B are simulated first on the two available processors and Node B reaches T_{S1} at T_{W0}. Similar to the case in Figure 1.1 of Chapter 1, Node B has to wait at T_{S1} until the simulation time of both Node A and Node C reach T_{S1}. However, unlike the case in Figure 1.1, while Node B is waiting, the simulation of Node C begins and both processors are kept busy. With the AEAP synchronization scheme, Node A should send its simulation time to Node B at T_{W1} so the simulation of Node B can be resumed. However, since both processors are busy simulating Node A and C at T_{W1}, reviving Node B at T_{W1} does not increase simulation performance at all. In fact, this may actually slow the simulation down due to the overhead in performing this unnecessary clock synchronization. For example, instead of synchronizing with Node B at T_{W1}, Node A can delay the synchronization until a free processor becomes available at T_{W2} when Node A needs to read the wireless channel and waits for Node B and C. By delaying the clock synchronization to T_{S2} at T_{W2}, Node A effectively reduces one clock synchronization.

Another area that the existing AEAP synchronization algorithms fail to exploit for synchronization reductions is the simulation time gaps among neighboring nodes [TLP05, WWM07, JG08]. Due to the lack of processors to simulate all non-waiting nodes simultaneously, the potential simulation time gaps of different nodes can be quite large during a simulation. For example, an actively transmitting node cannot hear transmissions from other nodes and therefore can be simulated without waiting until it stops transmitting and reads the wireless channel. Given such time gaps, a node

receiving the simulation time of a node in the future can compare the future node's time with its own simulation time and calculate potential dependencies between the two nodes in the future. Consequently, the node falling behind can skip clock synchronizations if there are no dependencies between the two nodes. For instance, as shown in Figure 5.1, once Node A sends a clock synchronization message to Node B at T_{S2}, Node B knows implicitly that Node A does not depend on it before T_{S2} and therefore does not need to synchronize its clock with Node A until then. In other words, Node B no longer needs to send its simulation time to Node A every minimal lookahead time before T_{S2} as it does with the AEAP synchronization algorithms. By delaying clock synchronizations, we can fully extend the time gaps and as a result create more opportunities for nodes falling behind to act upon and reduce clock synchronizations. We will discuss this in detail in the following section.

5.1.2 Lazy Synchronization Algorithm

To address the performance issue of the AEAP synchronization scheme, we propose a novel conservative synchronization scheme: LazySync. The key idea of the LazySync scheme is to delay a synchronization even when it should be done according to conservative simulations. It is opposite of opportunistic synchronization in that the simulator seeks to avoid synchronization until it is essential and it is able to do it given simulation resource constraints. Together, we show that the concept of lazy evaluation can be extended to specifically benefit from the operational characteristics of sensor networks. By procrastinating synchronizations, delayed clock synchronizations may be safely discarded or substituted by newer clock synchronizations in simulating WSNs. As a result, the total number of clock synchronizations in a simulation can be reduced.

Note that if free processors are available, our LazySync scheme must perform synchronizations AEAP so potential nodes can be revived to use the available physical resources. To make this possible, we track the number of non-waiting nodes and only procrastinate synchronizations when the number is below a threshold. Ideally, the threshold should be set to be the number of processors used to run the simulation in order to maximize clock synchronization reduction and processor usage. However, considering the frequency of checking the number of non-waiting nodes and the overheads in reviving waiting nodes and performing scheduling, the threshold should be set to a number slightly larger than that in practice. Tracking the number of non-waiting nodes

on a computer should incur very little overhead since that is already done by the underlying thread/process library or OS as part of their scheduling functions. For distributed simulations on multiple computers, the number of non-waiting nodes on each computer can be exchanged as part of clock synchronization messages sent between computers. If a computer does not receive any clock synchronization messages from another computer for a predetermined period of time, the nodes on the first computer can revert back to the AEAP scheme.

Algorithm 2: Lazy Synchronization Algorithm

Require: $syncThreshold$ /*sync threshold*/
Require: ΔT /*minimal lookahead time, the lookahead time of a node in the dependent state*/

```
1: set timer to fire at every ΔT
2: syncTime ⇐ 0 /*the time a sync condition is verified*/
3: while simulation not end do
4:     simulate the next instruction
5:     if in independent state then
6:         if timer.fired then
7:             syncTime ⇐ current sim time
8:             if numLiveNode < syncThreshold then
9:                 send current sim time to all neighboring nodes not ΔT ahead
10:            end if
11:        end if
12:    else if in dependent state then
13:        if instruction needs to read the wireless channel then
14:            syncTime ⇐ current sim time
15:            if ((Condition 1) == true) then
16:                if numLiveNode < syncThreshold then
17:                    send current sim time to all neighboring nodes not ΔT ahead
18:                end if
19:            else
20:                send current sim time to all neighboring nodes not ΔT ahead
21:                wait until ((Condition 1) == true)
22:            end if
```

```
23:             read the wireless channel
24:         end if
25:     end if
26:     if syncTime - (current sim time) > ΔT then
27:         syncTime ⇐ current sim time
28:         if numLiveNode < syncThreshold then
29:             send current sim time to all neighboring nodes not ΔT ahead
30:         end if
31:     end if
32: end while
```

Condition 1 *If a node N_i reads wireless channel C_k at simulation time T_{SN_i}, then for all nodes N_s that are in direct communication range of N_i, $(T_{SN_s} + \Delta T) \geq T_{SN_i}$, where T_{SN_s} is the simulation time of N_s and ΔT is the lookahead time of N_i which is in the dependent state.*

Our proposed LazySync algorithm is presented in Algorithm 2. As shown in Algorithm 2, we design the LazySync algorithm to work differently on nodes in different states because nodes may have different synchronization needs. In a simulation, a sensor node can be in one of two states, the independent state and the dependent state.

A node is in the independent state if its radio is not in receiving mode. This happens when the radio is off, in transmission mode or in any one of the initialization and transition states. Since a node in the independent state (independent node) does not take inputs from any other nodes, it can be simulated without waiting for any other nodes until the state changes. However, if free processors are available, an independent node still needs to synchronize with neighboring nodes so that the nodes depending on the outputs of the independent node can be simulated. In the LazySync algorithm, an independent node checks the number of non-waiting nodes every minimal lookahead time and only sends a clock synchronization message to its neighboring nodes if the number of non-waiting nodes is below a threshold.

A node is in the dependent state if its radio is in receiving mode. Since any node in direct communication range of a dependent node (a node in the dependent state) can potentially transmit, a dependent node needs to meet Condition 1 before actually reading the wireless channel to ensure correct simulation results. In other words, a dependent node needs to evaluate Condition 1 to determine if it can read the wireless

channel and continue the simulation or has to wait for some neighboring nodes to catch up for their potential outputs. Since a dependent node has its radio in receiving mode, it needs to read the wireless channel at least once every minimal lookahead time (ΔT) which is the lookahead time of a node in the dependent state. Therefore, Condition 1 is evaluated at least once every ΔT. In the LazySync algorithm, a dependent node only performs clock synchronizations under two circumstances. The first circumstance happens when Condition 1 is evaluated to be false and as a result, a dependent node has to wait for neighboring nodes. To prevent deadlocks, a synchronization has to be performed in this case before suspending the node, regardless of the number of available processors. A deadlock occurs when nodes wait for each other at the same simulation time. For instance, it happens when nodes within direct communication range read the wireless channel at the same simulation time. The second circumstance occurs when Condition 1 is evaluated to be true so a dependent node can go ahead to read the wireless channel. If the number of non-waiting nodes is below a threshold at this point, a clock synchronization is required to revive some nodes to use the available processors. Note that the block of code from line 26 to 31 in Algorithm 2 is just a safety mechanism to guard against the cases that a node does not stay in any of the two states long enough to check for synchronization conditions.

It is important to note that a dependent node may only perform clock synchronizations at the times it reads the wireless channel. This is very different from the case in a typical AEAP synchronization algorithm. A node in an AEAP synchronization algorithm may perform clock synchronizations at any time according to the waiting times of other nodes. The decision to limit dependent nodes to perform clock synchronizations at channel read time only is based on the assumption that there are no free processors available to simulate any other nodes until an actively running dependent node gives up its processor due to waiting. By procrastinating clock synchronizations to channel read time, we can eliminate all intermediate synchronizations that need to be performed otherwise in AEAP synchronization algorithms, as described in Section 5.1.1.

With the LazySync algorithm described above, a node can be simulated for a long period of time without sending its simulation time to neighboring nodes. As discussed in Section 5.1.1, the extended simulation time gaps of neighboring nodes can be exploited effectively to reduce clock synchronizations. According to Condition 1, a dependent node N_i can read the wireless channel only if the simulation time of all neighboring nodes are

equal to or greater than the simulation time of N_i minus ΔT. If the simulation time of a neighboring N_s is more than ΔT ahead of the simulation time of N_i, there are no needs for N_i to send its simulation time to N_s until the simulation time of N_i is greater than $T_{SN_s} - \Delta T$. The same also applies if an independent node receives a simulation time that is more than ΔT ahead. Based on these, the LazySync algorithm uses a filter to remove unnecessary clock synchronizations.

It is important to see that our LazySync algorithm still follows the principles of conservative synchronization algorithms [CM81, Fuj99a, RAF$^+$04] to not violate any causality during simulations. We only delay and discard unnecessary clock synchronizations to improve the performance of distributed simulations of WSNs. Due to space limits, a formal correctness proof of the LazySync algorithm is not given here.

5.2 Implementation

The proposed LazySync scheme is implemented in PolarLite, a distributed simulation framework that we developed based on Avrora as described in Chapter 3 [JG08]. As with Avrora, PolarLite allocates one thread for each simulated node and relies on the Java virtual machine (JVM) to assign runnable threads to any available processors on an SMP computer. However, we cannot identify any Java APIs that allow us to check the number of suspended/blocked threads in a running program. As an alternative, we track that using an atomic variable. The *syncThreshold* in Algorithm 2 is configurable via a command line argument.

To implement the LazySync algorithm, we need to detect the state that a node is in. In discrete event driven simulations, the changes of radio states are triggered by events and can be tracked. For example, in our framework, we detect the radio on/off time by tracking the IO events that access the registers of simulated radios. We verify the correctness of our implementation by running the same simulations with and without the LazySync algorithm using the same random seeds.

5.3 Evaluation

To evaluate the performance of the LazySync scheme, we simulate some typical WSNs with PolarLite using both the AEAP synchronization algorithm from [JG08] and the LazySync algorithm from Section 5.1.2. The performance results are compared

according to three criteria:

- $Speed_{avg}$: The average simulation speed.
- $Sync_{avg}$: The average number of clock synchronizations per node.
- $Wait_{avg}$: The average number of waits per node.

$Speed_{avg}$ is calculated using Equation (5.1). Note that the numerator of Equation (5.1) is the total simulation time in units of clock cycles. $Sync_{avg}$ is equal to the total number of clock synchronizations in a simulation divided by the total number of nodes in the simulation. Similarly, $Wait_{avg}$ is equal to the total number of times that nodes are suspended in a simulation due to waiting divided by the total number of nodes in the simulation.

$$Speed_{avg} = \frac{total\ number\ of\ clock\ cycles\ executed\ by\ the\ sensor\ nodes}{(simulation\ execution\ time) \times (number\ of\ sensor\ nodes)} \tag{5.1}$$

The WSNs we simulate in this section consist of only Mica2 nodes [Cro08] running either CountSend (sender) or CountReceive (receiver) programs. Both programs are from the TinyOS 1.1 distribution and are similar to the programs used by other WSN simulators in evaluating their performance [LLWC03, TLP05, WWM07]. For example, CountSend broadcasts a continuously increasing counter repeatedly at a fixed interval. If the interval is set to 250ms, it behaves exactly the same as CntToRfm which is used in [LLWC03, TLP05, WWM07] for performance evaluations. CountReceive listens for messages sent by CountSend and displays the received values on LEDS. All simulation experiments are conducted on an SMP server running Linux 2.6.24. The server features a total of 8 cores on 2 Intel Xeon 3.0GHz CPUs and 16GBytes of RAM. Sun's Java 1.6.0 is used to run all experiments. In the simulations, the starting time of each node is randomly selected between 0 and 1 second of simulation time to avoid any artificial time locks. All simulations are run for 120 seconds of simulation time and for each experiment we take the average of three runs as the results. The synchronization threshold (Algorithm 2) of the LazySync algorithm is set to 9 (the number of processors plus one) for all experiments.

5.3.1 Performance in One-hop WSNs

In this section, we evaluate the performance of the LazySync scheme in simulating one-hop WSNs of various sizes. One-hop WSNs are sensor networks with all their nodes

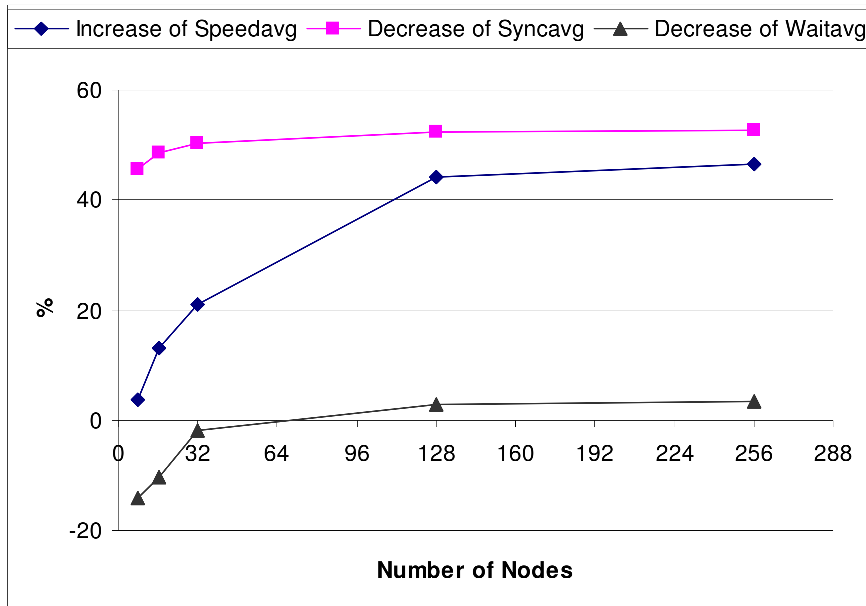

Figure 5.2: Performance improvements of the LazySync scheme over the AEAP scheme in simulating one-hop WSNs. Senders transmit at a 250ms interval.

in direct communication range. All the one-hop WSNs that we simulate in this section have 50% of the nodes running CountSend and 50% of the nodes running CountReceive.

In the first experiment, we modify CountSend so that all senders transmit at a fixed interval of 250ms. Five WSNs with 8, 16, 32, 128 and 256 nodes are simulated and Figure 5.2 shows the percentage improvements of the LazySync scheme compared to the AEAP scheme. As shown in Figure 5.2, the LazySync scheme reduces $Sync_{avg}$ in all cases and the percentage reductions grow slowly with network sizes. It is important to see that the total number of clock synchronizations in a distributed simulation of a one-hop WSN is on the order of $N * (N - 1)$ where N is the network size [JG08]. So, although the percentage reductions of $Sync_{avg}$ increase slowly with network sizes in Figure 5.2, the actual values of $Sync_{avg}$ decrease significantly with network sizes.

The significant percentage reduction of $Sync_{avg}$ in simulating 8 nodes with 8 processors is due to the time gap based filter and the fact that the synchronization threshold is only checked every ΔT or at channel read time. Since the threshold is not monitored at a finer time granularity, a processor may be left idle for a maximum of the amount of wallclock time to simulate a node for ΔT according to Algorithm 2. As a result, we can see in Figure 5.2 that there are moderate increases of $Wait_{avg}$ when

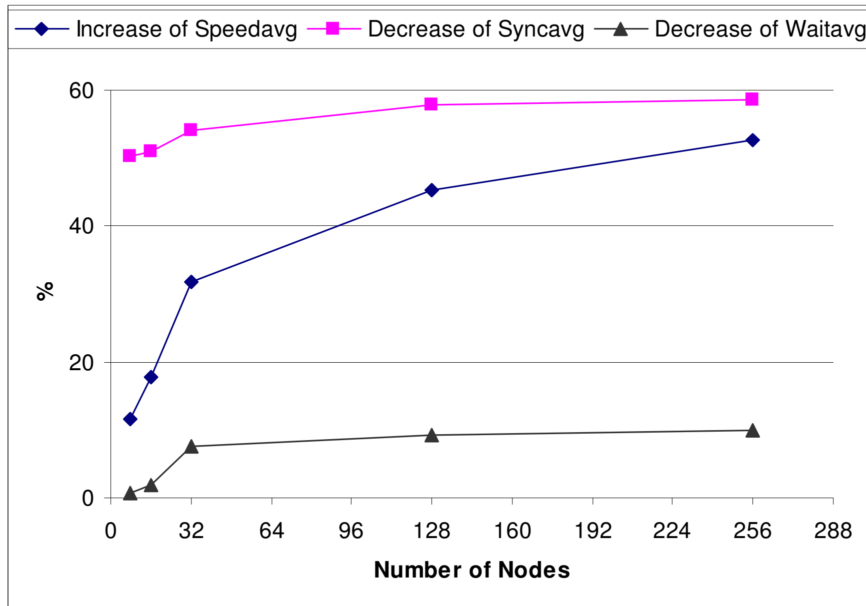

Figure 5.3: Performance improvements of the LazySync scheme over the AEAP scheme in simulating one-hop WSNs. Senders transmit as fast as possible.

simulating small WSNs with 8 and 16 nodes. However, as the WSN size increases, the percentage reduction of $Wait_{avg}$ increases because processors are more likely to be kept busy by the extra nodes. In fact, the LazySync scheme performs better in terms of percentage reductions of $Wait_{avg}$ when simulating 128 and 256 nodes, as shown in Figure 5.2. We believe this is because more CPU cycles become available for real simulations after significant reductions in the number of clock synchronizations. For the same reason, despite the increases of $Wait_{avg}$ in simulating small WSNs, we see increases of $Speed_{avg}$ in all cases, ranging from 4% to 46%.

Our second experiment is designed to evaluate the LazySync scheme in busy WSNs that have heavy communication traffic. It is based on the same setup as the first experiment except all senders transmit as fast as possible. As shown in Figure 5.3, the LazySync scheme provides more significant percentage reductions of $Wait_{avg}$ in busier networks. This is because a busier network has more transmissions and consequently more independent states. The increased number of independent states in a busier network provides more opportunities for the LazySync scheme to exploit. It allows nodes to skip synchronizations and gives the filter larger gaps to exploit. As a result, the LazySync scheme brings a 12% to 53% increase of $Speed_{avg}$ in Figure 5.3. We can also see in

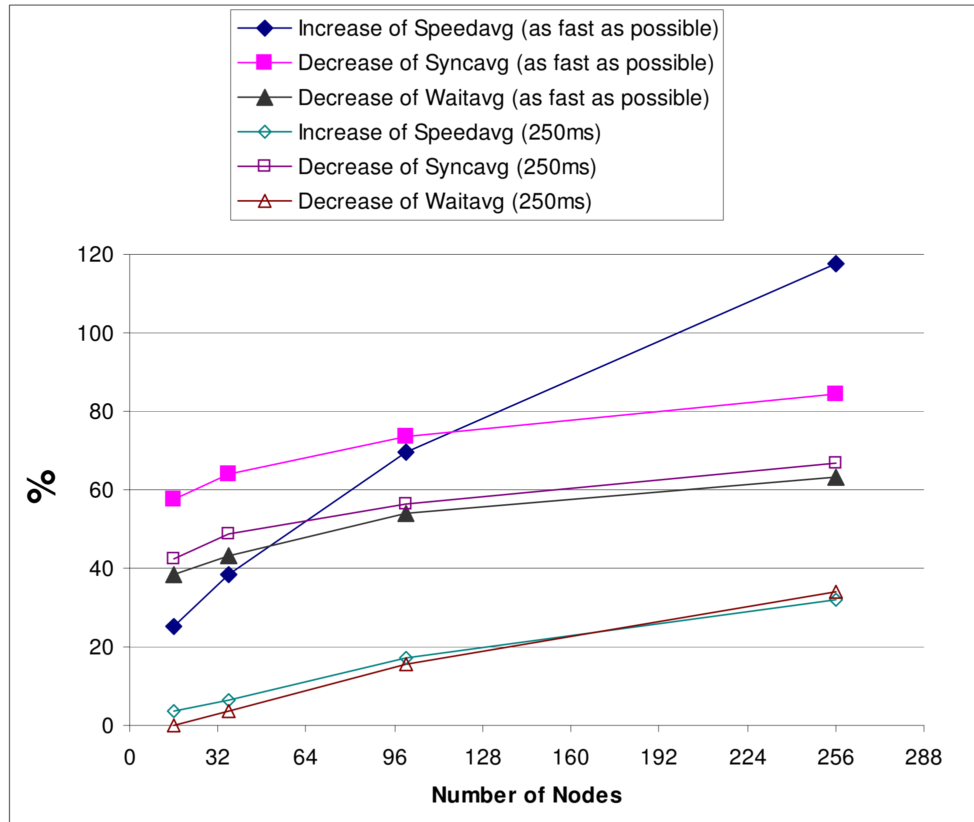

Figure 5.4: Performance improvements of the LazySync scheme over the AEAP scheme in simulating multi-hop WSNs.

Figure 5.3 that there are no increases of $Wait_{avg}$ in simulating small WSNs as in the first experiment. This is because it takes more CPU cycles to simulate all the communications in a busy network and that keeps the processors busy.

5.3.2 Performance in Multi-hop WSNs

In this section, we evaluate the performance of the LazySync scheme in simulating multi-hop WSNs of various sizes. Nodes are laid 15 meters apart on square grids of various sizes. Senders and receivers are positioned on the grids in such a way that nodes of the same types are not adjacent to each other. By setting a maximum transmission range of 20 meters, this setup ensures that only adjacent nodes are within direct communication range of each other. This configuration is very similar to the two dimensional topology in DiSenS [WWM07].

We simulate WSNs with 16, 36, 100 and 256 nodes. For each network size, we

simulate both a quiet network with all the senders transmitting at a fixed 250ms interval and a busy network with all the senders transmitting as fast as possible. The results are shown in Figure 5.4. We can see that the percentage decreases of $Sync_{avg}$ are more significant in the multi-hop networks than in the one-hop networks. The reason for this is that there are fewer dependencies among nodes in our multi-hop networks than in the one-hop networks, as a result of only having adjacent nodes in communication range in the multi-hop network setup. Having fewer dependencies brings two opportunities to the LazySync scheme. First, a node can be simulated for a longer period of time without waiting. Second, the increased number of non-waiting nodes keeps processors busy. Together, they enable nodes to skip clock synchronizations in LazySync. In addition, the increased simulation time gaps can also be exploited by LazySync to reduce clock synchronizations. As shown in Figure 5.4, the percentage reductions of $Sync_{avg}$ are significantly higher in the busy multi-hop networks than in the quiet ones. This demonstrates once again that the LazySync scheme can exploit wireless transmissions in a WSN for synchronization reductions. As a result, we see significant percentage increases of $Speed_{avg}$ in simulating busy multi-hop networks, ranging from 25% to 118%.

5.4 Related Work

Increasing lookahead time [CM81] is a commonly used approach [FKM92, LN02] to improve the performance of conservative simulators. Our previous work in Chapter 4 and 5 follows this direction. LazySync is very different from the previous approaches because it works by identifying and eliminating unnecessary synchronizations during simulations. In addition, LazySync is particularly effective in improving the performance of simulating dense WSNs with a large amount of communication traffic.

LazySync is based on similar ideas as lazy evaluations which have been used in earlier work on very different problems from architectural designs to programming languages [Hug89]. Though conceptually similar, we show how the notion of lazy evaluation can be applied in sensor networks for improved performance of distributed simulations.

5.5 Summary

We have presented LazySync, a synchronization scheme that significantly improves the speed and scalability of distributed sensor network simulators by reducing

the number of clock synchronizations. We implemented LazySync in PolarLite and evaluated it against an AEAP scheme inside the same simulation framework. The significant improvements of simulation performance on a multi-processor computer in our experiments suggest even greater benefits in applying our techniques to distributed simulations over a network of computers because of their large overheads in sending synchronization messages across computers during simulations.

Acknowledgements: Chapter 5, in part, has been published as "A New Synchronization Scheme for Distributed Simulation of Sensor Networks" by Zhong-Yi Jin and Rajesh Gupta in DCOSS 09: Proceedings of the 5th ACM/IEEE International Conference on Distributed Computing in Sensor Systems [JG09b], pages 103-116. The dissertation author was the primary investigator and author of this paper.

Chapter 6

A Framework for Evaluating the Performance of Conservative and Optimistic Simulations of WSNs

In this chapter, we study the relative performance of the conservative and optimistic approaches in simulating WSNs. We describe a technique that evaluates the performance of conservative and optimistic approaches accurately using simulation traces. Simulation traces can be collected from actual simulations using any distributed or sequential simulators. The technique is independent of the trace collection process and separates the simulation overheads from the actual simulation algorithms. This makes it possible to use the same traces to quickly prototype and study both approaches with any design tradeoffs, speedup techniques and optimizations. In addition, the technique can evaluate the performance of both approaches on virtual computing platforms with arbitrary numbers of CPUs without using real hardware. With this technique, we develop the SimVal simulation performance evaluation framework and evaluate the relative performance of the two approaches in simulating some typical WSNs.

6.1 Introduction

The fundamental design decision in building parallel and distributed WSN simulators relates to how to keep sensor nodes synchronized during simulations while providing good simulation speed and scalability. The conservative and optimistic approaches

work very differently in preserving the causality of events [TLP05, WWM07, JG08] and therefore affect simulation performance differently in terms of simulation speed and scalability. A significant amount of research [ACD+92, Nic93, TTA98, Fuj99b, Son01], both experimental and analytical, has been conducted to study the performance of the conservative and optimistic approaches on many types of applications. Results show that the relative performance of the two approaches depends largely on the characteristics of the applications under simulation. In other words, a better performance in simulating a queuing network with the optimistic approach does not necessarily suggest it can also provide a better performance in simulating an air traffic control system.

Given that it is difficult to estimate the performance of the approaches for a specific application due to the fundamental differences in how events are handled with these two approaches, rough guidelines based on look-ahead time, communication patterns, and simulation overheads are proposed to help choose the right approach for the application [Fuj99b]. However, the various design tradeoffs, numerous speedup techniques, and subtle differences in different simulation scenarios make it unreliable to use the rough guidelines to predict the relative performance of the two approaches for a given application. In the case of simulating WSNs, there are significant amounts of diversity in WSNs and their applications. Compared to pure network level simulations, the need for accurate simulations of low level details such as node energy usage further increases the complexity and diversity of WSN simulations. As a result, there are no existing guidelines or ways to tell which approach is more effective in simulating a WSN application.

In this chapter, we focus on studying the performance of the conservative and optimistic approaches in simulating WSNs. For a given simulation, the performance of the two approaches can be greatly affected by tradeoffs in controlling simulation overheads, speedup techniques and how well the simulation models are optimized for the approaches. A direct performance comparison of the two approaches is difficult since one has to develop two very different simulators with different design tradeoffs, speedup techniques and simulation models. This is especially challenging for WSNs since we cannot find any existing optimistic WSN simulators and it is more difficult to implement an optimistic simulator than a conservative one due to inherent complexity. In addition, a simulator is usually implemented and optimized for a specific hardware architecture such as symmetric multiprocessors (SMPs) or clusters, making it challenging

to compare simulation performance over different hardware architectures. Finally, direct comparisons are limited by available hardware and that makes it hard to study the scalability of the approaches on very large numbers of CPUs or on future generations of CPUs that feature large numbers of cores. Performance evaluations based on analytical methods are also challenging due to the complexity of the simulation models and the intricate interactions of a large number of sensor nodes. To address these challenges, we develop the SimVal framework that evaluates the performance of both approaches accurately using simulation traces.

6.2 Idea and Approach

There are two notions of time in a simulation. *Simulation time* is the virtual clock time in the simulated models [Fuj99a] while *wallclock time* corresponds to the actual physical time used in running the simulation program. Regardless of what type of distributed simulation approach is used, the amount of wallclock time consumed by a simulation can always be divided into three parts:

- Resource-contention-time: Those times used to wait for the availability of simulation resources such as processors.
- Causality-preservation-time: Those times used to preserve causality relations in the simulation.
- Real-work-time: Those times used to perform the actual simulation work.

The first two parts are essentially the overheads of the simulation and the third part determines the upper bound on the simulation speed which is defined as the ratio of simulation time over wallclock time. For example, Figure 6.1 shows the progress of simulating with the conservative approach a sensor network of 2 nodes on a single CPU. We can see that Node B is simulated first and the simulation of Node B is paused at wallclock time $T_{W\alpha}$. This is because according to the conservative approach, before Node B can read the wireless channel at simulation time T_{S1}, it needs to wait for Node A at that point to ensure no causality is violated. Since there is only one CPU available for the simulation, Node A cannot be simulated until the simulation of Node B is paused at wallclock time $T_{W\alpha}$. In other words, $T_{W\alpha}$ is a part of the resource-contention-time of

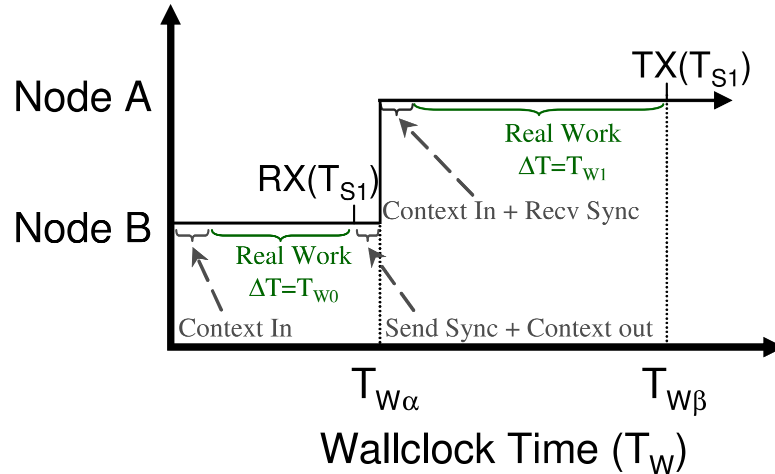

Figure 6.1: The progress of simulating with the conservative approach on one CPU a wireless sensor network of two nodes that are in direct communication range of each other.

Node A. Similarly, the overhead time of context in/out the thread or process simulating Node A or B is counted as a part of the resource-contention-time of the corresponding node as well. With the conservative approach, Node B must synchronize with Node A before the waiting at T_{S1} to prevent deadlocks. The overhead time of sending the synchronization message is counted as a part of the causality-preservation-time of Node B. Similarly, the time that it takes for Node A to process the synchronization message is a part of the causality-preservation-time for Node A. Other overheads such as those from scheduling the threads/processes are not shown in Figure 6.1 for simplicity. They are described in detail in Section 6.4. The real-work-times for Node A and B are T_{W1} and T_{W0} respectively, as shown in Figure 6.1.

The basic idea of SimVal is to evaluate the performance of the conservative and optimistic approaches using ideal-traces. Ideally, if there are no overheads in a simulation (both resource-contention-time and causality-preservation-time are zero) and we log, in the order they occur, every safe event of every node in the simulation, both conservative and optimistic simulators would generate the exact same sequence of event logs for each node and we call such sequences of event logs the *ideal-trace* of a simulation. Figure 6.2 shows the ideal-trace of the simulation scenario illustrated in Figure 6.1.

In principle, an ideal-trace contains both the information about the causality of events as well as the actual real-work-time that it takes to generate each event in a

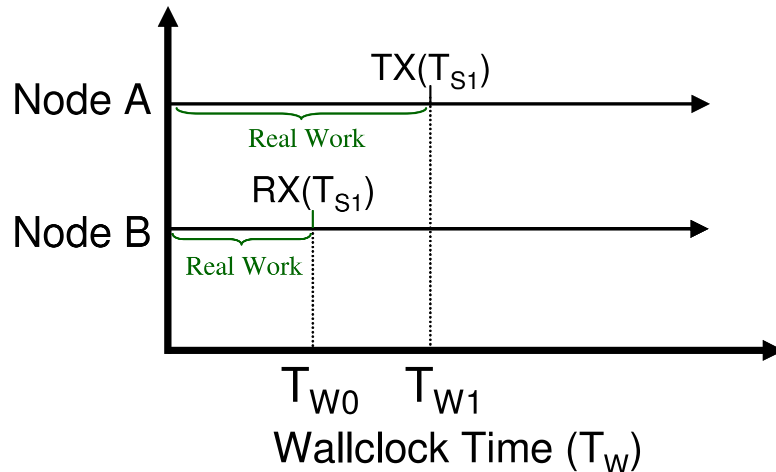

Figure 6.2: The ideal-trace of the simulation scenario in Figure 6.1.

real simulation. Therefore, we can evaluate the performance of different simulation approaches and algorithms by replaying the events and introducing the resource-contention-time and causality-preservation-time piece by piece during the playback process as if in a real simulation. In addition, the events can be played back on any numbers of virtual CPUs so simulation performance on any numbers of CPUs can be evaluated without using real hardware. For example, we can use the ideal-trace in Figure 6.2 to play back the scenario in Figure 6.1 on a single CPU using the optimistic approach. If we pick Node B to play back first, it has to be context switched in and the context-in overhead is added into the playback process. After advancing to T_{S1}, Node B would have to save the simulation state at T_{S1} and the state saving overhead is added into the playback process. Similarly, when Node A is advanced to T_{S1}, Node B needs to roll back to T_{S1} and the rollback overhead is added into the playback process. The playback process is discussed in detail in Section 6.3. As shown in Figure 6.2, every event in an ideal-trace is represented by a 4-tuple of ⟨Node ID, Event Type, Wallclock Time, Simulation Time⟩. For example, the RX event of Node B in Figure 6.2 is represented by ⟨Node-B, RX, T_{W0}, T_{S1}⟩.

It is important to note that the playback process in SimVal is significantly simpler than performing a real simulation because for SimVal the actual simulation work is already done and the results are represented as events in the ideal-trace. For instance, the transmission of Node A in Figure 6.1 is represented by the TX event in the ideal-trace of Figure 6.2. SimVal does not need to know why there is a TX event at T_{S1} since

it is the output of the original simulation from which the ideal-trace is constructed. In addition, it is significantly easier to use SimVal to evaluate different tradeoffs and speedup techniques for distributed simulations because we can change their effects on resource-contention-time and causality-preservation-time easily during the playback process. For instance, a commonly used technique to reduce state saving overhead and memory usage in optimistic simulations is to save the incremental changes from the previous saved state rather than the entire new state. However this increases the rollback overhead since the state to rollback to has to be recovered from the incremental changes. With SimVal, we can quickly prototype and study this tradeoff by just changing the corresponding overhead numbers in the optimistic playback process. Similarly, we can use SimVal to evaluate simulation performance over different hardware architectures such as SMPs or clusters.

6.2.1 Ideal-Trace

Ideal-traces cannot be collected directly from an actual simulation because unavoidably all simulation approaches would introduce overhead times due to resource-contention or causality-preservation. However, we can construct an ideal-trace from a normal simulation trace by removing the resource-contention-time as well as the causality-preservation-time from the wallclock time of the events in the normal trace. For example, a normal trace for simulating the scenario in Figure 1.1 can be collected by logging the simulation events of each node as they occur in an actual simulation using the conservative approach. From this normal trace, we know that Node B waits for Node A from T_{w0} to T_{w1} and that time can be removed from the wallclock time of all the future events of Node B when building the ideal-trace. Normal traces can be collected using any distributed or sequential simulation approaches as long as we can keep track of the resource-contention-time and the causality-preservation-time in a simulation. For experiments in this chapter, normal traces are collected with distributed PolarLite simulator [JG08, JG09a]. However, with appropriate loggings, we could also collect normal traces using sequential simulators such as TOSSIM [LLWC03]. This is discussed in detail in Section 6.4.

Only certain types of events are required in an ideal-trace for evaluating the performance of the conservative and optimistic approaches in simulating WSNs. This is because for WSNs, nodes only interact with each other via radio messages. In other

words, causality only occurs when a node reads from or writes to the wireless channel. Therefore, for evaluating WSN simulators, we only need to log channel read (RX), channel write (TX) and channel switch events in a simulation run besides keeping track of events that correspond to resource-contention-time and causality-preservation-time.

6.3 SimVal

The core of SimVal is a scheduler that drives the event playback process. The SimVal playback-scheduler works in two separate steps: schedule nodes onto virtual CPUs and schedule events of the nodes running on the virtual CPUs. A virtual CPU has no correlations to the CPUs on which SimVal is executed. It is just a programming construct in the SimVal scheduler that represents a CPU on which the sensor nodes are simulated. The number of virtual CPUs is specified before a playback starts and the nodes and events are scheduled the same way as in a real simulation. When scheduled events are processed, overhead times due to resource-contention and causality-preservation are introduced and new events are created according to the specific simulation approach and algorithms under evaluation. Once all currently scheduled events are processed, the SimVal playback-scheduler is invoked again to schedule the future events, including the newly created ones, for processing. This process continues until all events are scheduled and processed. We describe how the SimVal playback-scheduler works for the conservative and optimistic approaches separately in Sections 6.3.1 and 6.3.2 using the following notations:

- W: Wallclock time. $W10$ means at wallclock time 10.
- S: Simulation time. $S10$ means at simulation time 10.
- O: Overhead time from resource-contention or causality-preservation. $O10$ means 10 units of overhead time.
- $N(w, s)$: Node N at wallclock time w and simulation time s.
- $N(w, s, o)$: Node N at wallclock time w, simulation time s and with o units of overhead time to be introduced.

6.3.1 Conservative Playback

The conservative scheduling algorithm of SimVal is presented in Algorithm 3. With the conservative simulation approach, a simulated node can be in one of four states: running, waiting, runnable and terminated. A node is in the running state if it is actively being simulated on a CPU. A waiting node is a node that is waiting for outputs from other nodes due to the causality constraints. In other words, a waiting node cannot be simulated any further until the conditions that it is waiting for are satisfied. A runnable node is a node that is not in the waiting state but cannot be simulated due to the lack of physical resources such as CPUs. A terminated node is a node that no longer needs to be simulated.

Algorithm 3: SimVal Conservative Playback-Scheduler

Require: nl /*a list of nodes*/
Require: cl /*a list of virtual CPUs*/
1: **while** ∃ unterminated nodes in nl **do**
2: **for** every $freeCPU$ in cl **do**
3: **if** ∃ a $runnableNode$ in nl **then**
4: Schedule $runnableNode$ to $freeCPU$
5: **end if**
6: **end for**
7: /*Find the events to schedule next*/
8: $st = MIN$ /*set schedTime to the min*/
9: **for** every running node $N(w_{cur}, s_{cur}, o_{cur}) \rightarrow (w_{next}, s_{next}, o_{next})$ in nl **do**
10: **if** $st < (w_{next} - w_{cur} + o_{cur})$ **then**
11: $st = w_{next} - w_{cur} + o_{cur}$
12: **end if**
13: **end for**
14: /*Advance running nodes and schedule events*/
15: **for** every running node $N(w_{cur}, s_{cur}, o_{cur}) \rightarrow (w_{next}, s_{next}, o_{next})$ in nl **do**
16: **if** $st == (w_{next} - w_{cur} + o_{cur})$ **then**
17: Add events at $N(w_{next}, s_{next}, o_{next})$ to N.processQueue
18: **else**

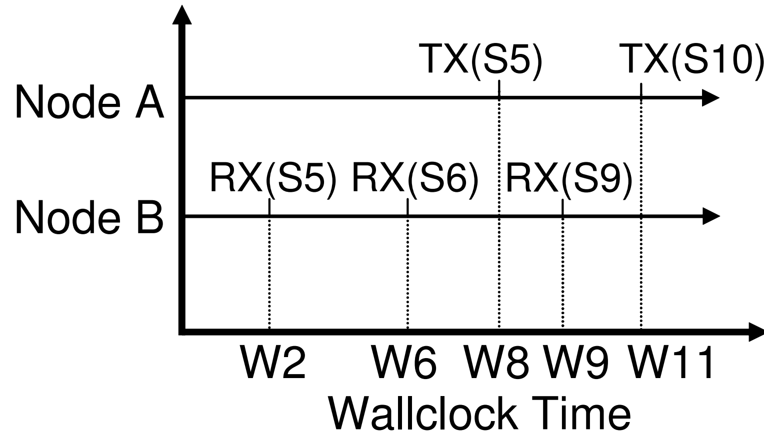

Figure 6.3: The trace of simulating a wireless sensor network with two nodes that are in direct communication range of each other.

19: Advance N to
20: $N(w_{cur} + st, s_{cur}, o_{cur}) \rightarrow (w_{next}, s_{next}, o_{next})$
21: **end if**
22: **end for**
23: /*Process scheduled events*/
24: **for** every running node N with scheduled events **do**
25: **for** every event e in $N.processQueue$ **do**
26: Advance N to e
27: Process e
28: **end for**
29: **end for**
30: **end while**

Similar to the scheduler in a conservative simulator, the SimVal conservative playback-scheduler assigns one runnable node to each virtual CPU at the beginning of the playback process. After that, it performs a new assignment whenever a node enters into the waiting or terminated state and there are runnable nodes left. The specific order on which the nodes are assigned to the virtual CPUs depends on the specific scheduling policy under evaluation [SMB00]. Once nodes are assigned to the virtual CPUs, the events of the running nodes are scheduled and processed. The details of how the events are scheduled and processed can be best described using the so-called playback-state.

The playback-state of a simulated Node N consists of two sub-states: the current state $N(w_{cur}, s_{cur}, o_{cur})$ and the next state $N(w_{next}, s_{next}, o_{next})$. The current state is the latest state of a node in the playback process while the next state is where the node will advance to next. The states are driven by the events in the input ideal-trace as well as new events created during the playback. For example, if we denote a playback-state with $N(w_{cur}, s_{cur}, o_{cur}) \rightarrow N(w_{next}, s_{next}, o_{next})$, then at the beginning of playing back the ideal-trace in Figure 6.3, the initial playback-states of nodes A and B are $A(0,0,0) \rightarrow A(8,5,0)$ and $B(0,0,0) \rightarrow B(2,5,0)$ respectively. $A(0,0,0)$ and $B(0,0,0)$ are the default beginning states while $A(8,5,0)$ and $B(2,5,0)$ represent the first events of the two nodes in Figure 6.3 and are where the nodes will advance to next.

Once the initial playback-states are established, the events to schedule next are determined by the playback-states of the running nodes that have the minimum $advanceTime$, which is defined as $w_{next} - w_{cur} + o_{cur}$, among all running nodes. In other words, the earliest events of the running nodes are scheduled next and multiple events can be scheduled at the same time. It is important to note that the difference of the wallclock time $w_{next} - w_{cur}$ represents the actual time (wallclock) it takes in a real simulation to process the current event and advance to the next event. In other words, it represents the real-work-time as defined in Section 6.2. The wallclock times of events are never used alone because individually they carry no meaning in the context of playback. The o_{cur} part of $advanceTime$ is used to introduce overhead times and we will discuss that later in this section. This process is repeated to schedule remaining events once all currently scheduled events are processed.

Scheduled events are processed on line 27 of Algorithm 3. The event processing step injects simulation overheads into the playback process by creating new events according to the specific event under processing as well as the specific simulation approach and algorithms under evaluation. Since the event processing step is an integral part of the event scheduling process, we describe it along with the event scheduling steps using a concrete example.

For example, when we play back the ideal-trace in Figure 6.3 with a conservative playback-scheduler on two virtual CPUs, Nodes A and B are scheduled onto the CPUs as running nodes at first and the initial playback-states are initialized to $A(0,0,0) \rightarrow A(8,5,0)$ and $B(0,0,0) \rightarrow B(2,5,0)$ accordingly. Then according to the minimum $advanceTime$ which is 2, the RX event at $B(2,5)$ is scheduled first and the nodes

advance to $A(2,0,0)\rightarrow A(8,5,0)$ and $B(2,5,0)\rightarrow B(6,6,0)$ to process the RX event. According to the conservative approach, after processing the RX event at $B(2,5)$, Node B needs to synchronize with Node A (to avoid deadlock) by sending its simulation time to Node A and wait at $S5$ (to preserve causality) until the simulation time of Node A reaches $S5$. To model this, when the RX event is processed, the SimVal conservative playback process first injects the synchronization overhead time ($syncTime$), the time it takes for Node B to perform a synchronization, into the playback process. This is done by creating a new synchronization-completion event at $B(2+syncTime, 5, syncTime)$ and updating the playback-state of Node B to $B(2,5,0)\rightarrow B(2+syncTime, 5, syncTime)$.

Let us assume without loss of generality that $syncTime$ equals 1 unit of time, then the next event to schedule is at $B(3,5,1)$ and the running nodes advance to $A(3,0,0)\rightarrow A(8,5,0)$ and $B(3,5,1)\rightarrow B(6,6,0)$ to process the synchronization completion event of Node B. When the synchronization completion event is processed, the SimVal conservative playback process creates a new synchronization event and adds that to the event list of Node A. This new event contains the current simulation time of Node B and the current wallclock time of Node A so Node A can process the event later on as if it is in a real simulation. Additional overhead times and events could be introduced similarly, e.g., the overhead time that Node A needs to process a synchronization message. A complete list of overhead times supported by SimVal can be found in Section 6.4.

Assume, for ease of explanation, that no other overhead times are introduced and therefore Node B is put into the waiting state right away as a result of finishing processing the synchronization completion event at $B(3,5,1)$. Then the next event to schedule is the TX event at $A(8,5)$ and the running node advances to $A(8,5,0)\rightarrow A(11,10,0)$ while the waiting node remains at $B(3,5,1)\rightarrow B(6,6,0)$. After processing the TX event, let us assume again for ease of explanation that without any extra overheads, Node B is activated right away. Then the next event to schedule is the TX event at $A(11,10)$. The RX event at $B(6,6)$ is not scheduled because it has an $advanceTime$ of 4 due to the 1 unit of overhead time from performing the synchronization. The overhead time is crucial in calculating the $advanceTime$ as the physical time to reach the next event is delayed by that amount. To process the TX event at $A(11,10)$, the running nodes advance to $A(11,10,0)\rightarrow A(end, end, 0)$ and $B(6,5,1)\rightarrow B(6,6,0)$, and the simulation of Node A is terminated. After that, the active node will advance to $B(6,6,0)\rightarrow B(9,9,0)$. The overhead time is cleared once an event from the ideal-trace is processed. This is

because the next events to schedule are determined by the physical time difference from the current event rather than the absolute physical time of the next event. Finally, Node A advances to $B(9,9,0) \rightarrow B(end, end, 0)$ and terminates.

Once the entire playback process is over, we can find how much wallclock time has advanced for each node and the largest one is the estimated execution time. Execution time is the amount of time that it takes to complete the simulation using the specific simulation approach and algorithms under evaluation.

6.3.2 Optimistic Playback

The SimVal optimistic scheduling algorithm is similar to the conservative one in Algorithm 3. In fact, we could use the same playback-scheduler to evaluate the performance of the optimistic approach by changing the event processing step on line 27 of Algorithm 3 to introduce overhead times and new events according to the optimistic approach and algorithms. However, the optimistic playback scheduler generally needs to support a more sophisticated scheduling algorithm in assigning Nodes to CPUs. This is because with the optimistic approach, simulated nodes do not need to wait for each other due to causality constraints and therefore can only be in one of 3 states: running, runnable and terminated. If there are more nodes than the number of CPUs and some nodes are simulated first, it becomes important for performance reasons to prevent the simulation time of the nodes from falling far apart. Otherwise, incorrect paths can be simulated in places of correct ones causing large number of rollbacks. For instance, when we simulate the scenario in Figure 6.4 on two CPUs and pick Nodes A and B to simulate first, if Nodes A and B are simulated all the way to the end, both A and B will be rolled back to the very beginning once Node C is simulated to $C(2,4)$. A common scheduling policy to reduce such rollbacks is to simulate in a round robin fashion each node for a certain amount of events or time [Fuj99b].

Rollbacks can occur under two conditions in optimistic simulations of WSNs. The first condition is when a node receives a transmission from the past (in simulation time) and the node, in RX mode at the transmission time, did not guess correctly the occurrence or the content of the transmission. When that happens, the node may need to roll back to a saved state with a simulation time stamp that is less or equal to the simulation time of the transmission and this type of rollback is called *primary-rollback* [Fuj99b]. The second condition happens after a primary-rollback. If a node has

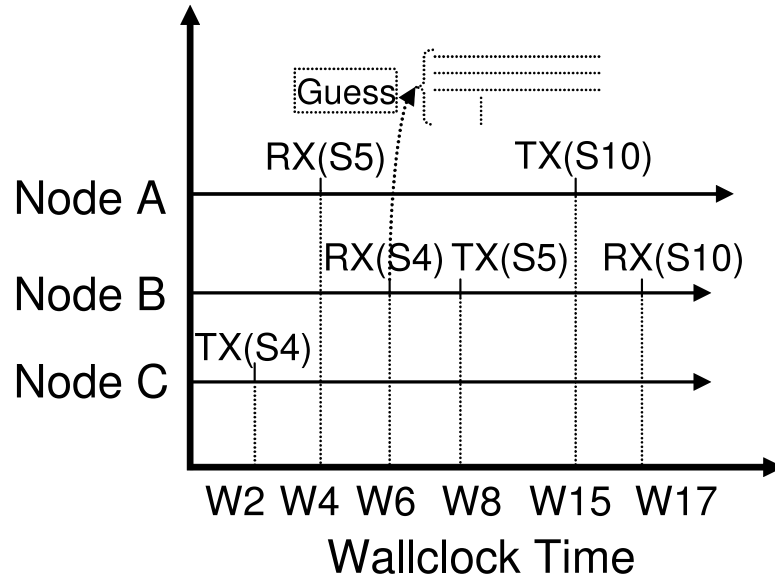

Figure 6.4: The ideal-trace of simulating a wireless sensor network with three nodes that are in direct communication range of each other.

made transmissions in the future of the rollback time, all other nodes that have received the transmissions may also need to roll back and this type of rollback is referred as *secondary-rollback* [Fuj99b]. In other words, if a node makes a wrong guess at time Sx, the node eventually has to roll back to Sx regardless of what path the node has taken and withdraw all the messages it has transmitted over that wrong path. This is the key that we can use the ideal-trace to evaluate the optimistic approach without knowing the wrong paths. For example, if we play back the trace of Figure 6.4 optimistically on 1 CPU and pick Nodes B to simulate first, Node B will make a guess at $B(6,4)$. If the guess at $B(6,4)$ is wrong, then Node B will have different internal states and follow a different path. However, regardless of what has happened over that path, once the transmission at $C(4,2)$ is received, Node B must roll back to $B(6,4)$ and withdraw all the transmissions it has made over the wrong path.

Transmissions or messages are typically withdrawn by sending anti-messages [Jef85]. An anti-message is identical to the message it is supposed to withdraw except it contains an extra field indicating that it is for cancellation purposes. After receiving an anti-message, a node either deletes the original message if it has not been processed or rolls back to a saved state with simulation time non-greater than the simulation time of the message that needs to be withdrawn.

The number of messages that need to be withdrawn after a wrong guess largely depends on the sensor node programs under simulation and what optimization techniques, such as *lazy cancellation* [Gaf88, Fuj99b], are used. It is important to note that the messages that need to be withdrawn come from simulation artifacts and a special mechanism has to be provided in an optimistic simulator to deal with unhandled errors of nodes when unexpected messages are received. Since a receiver of the messages only needs to roll back to one of the messages that has the smallest time stamp if a rollback is required, we abstract the message-withdraw-probability as an input to the SimVal playback process. Specifically, the message-withdraw-probability of a node is represented in SimVal as the probability that the node will withdraw a message one look-ahead time after the rollback time. This not only allows us to fine tune SimVal for different simulation scenarios but also enables us to evaluate any speedup techniques that effectively reduce the number of messages need to be withdrawn.

6.4 Implementation

The trace collection process is implemented in PolarLite [JG08, JG09a], a conservative distributed simulation framework developed based on the Avrora simulator [TLP05] as described in Chapter 3. PolarLite provides the same level of cycle accurate simulation as Avrora but uses a distributed synchronization engine instead of Avrora's centralized one.

Like any other discrete event driven simulators, PolarLite is driven by events and each event is processed by a handler (function) that is specific to that event. This makes trace collection straightforward since we only need to add logging code to the beginning or end of the handlers that process the events of interest. The exact same technique can be used to collect traces from any other event driven simulators such as TOSSIM [LLWC03]. TOSSIM is a sequential simulator but from the trace collection point of view, the only difference between PolarLite and TOSSIM is that events in PolarLite are dispatched from multiple event queues (one for each note) but from a single shared global queue in TOSSIM. For sequential simulators, we do not need to collect synchronization overheads since events are synchronized naturally by the shared global queue.

The types of simulation events collected from PolarLite as normal traces are listed in Table 6.1. Each event is represented in the traces by a 4-tuple as described in Section 6.2. The radio-TX/RX-on events in Table 6.1 are collected for evaluating certain

Table 6.1: Trace events

Event type	Description
Trace start/stop	Start and stop of a trace
Channel read/write	Channel access
Radio TX on/off	Radio in/out of TX mode
Radio RX on/off	Radio in/out of RX mode
Sync start/stop	The beginning and end of a synchronization
Wait start/stop	The beginning and end of a waiting period
Safe-time update start/stop	The beginning and end of updating the earliest input time

Table 6.2: Supported conservative overheads

Overhead Type	Description
Sync send overhead	Overhead time in sending a sync message
Sync receive overhead	Overhead time in processing a sync message
Context switch overhead	Overhead time of putting a running node into sleep and scheduling a waiting node to run
Safe-time update overhead	Overhead time of updating the earliest input time

speedup techniques [JG09a] but are not used in our current implementation. The safe-time-update events occur when a node needs to update the earliest input time which is the node's latest view of the earliest simulation time of the neighboring nodes. The safe-time is used to check if it is safe to read the wireless channel without causing any causality violations.

With trace-enabled PolarLite, normal simulation trace of each node is collected separately and kept in memory during a simulation. They are written to the disk in a single file once the simulation is completed. When a playback starts, we check if the input file contains a normal simulation trace and if so construct an ideal-trace from that.

Ideal-traces are constructed in SimVal by removing the synchronization and waiting time from the events in our normal traces. Note that resource-contention-time, which is a part of the waiting time, is calculated from the trace-start-time and wait-

Table 6.3: Supported optimistic overheads

Overhead Type	Description
Anti-Message send overhead	Overhead time in sending an anti-message
Anti-Message receive overhead	Overhead time in processing an anti-message
State saving overhead	Overhead time in performing a state saving
Rollback overhead	Overhead time in performing a rollback
Context switch overhead	Overhead time of putting a running node into sleep and scheduling a waiting node to run

start/stop-time in Table 6.1. The average overhead time of logging an event is measured experimentally and removed as well when the ideal-traces are constructed.

The playback process of SimVal is implemented in a separate program called AnTrace. AnTrace reads in an ideal-trace, performs the playback according to the specified inputs such as the type of simulation approach and the number of virtual CPUs and reports the estimated simulation speed once the playback is completed. In the current implementation, the types of overheads that are introduced into the SimVal playback process are listed in Table 6.2 and 6.3 for the conservative approach and the optimistic approach respectively. Note that we combine the context-out overhead, the scheduling overhead and the context-in overhead into the context switch overhead since the corresponding events almost always occur in sequence. For optimistic simulations, the safe-time is mainly used by some speedup techniques and we do not yet support the safe-time-update-overhead in the optimistic playback process. The current conservative scheduler for assigning nodes to virtual CPUs is implemented as in Algorithm 3. Its optimistic counterpart is implemented as a round-robin scheduler.

Besides supporting the classic conservative simulation protocol [CM81], we also implement in AnTrace a speedup technique called *LazySync* which can significantly improve the speed of conservative WSN simulators by reducing the number of synchronizations [JG09b]. This allows us to evaluate the performance of the conservative approach with or without a speedup technique. The implementation of the *LazySync* algorithm is straightforward in SimVal since SimVal is sequential by nature and therefore one does

not need to worry about the complexity of writing actual parallel programs. Running in sequential does not slow down SimVal significantly since by using ideal-trace, SimVal does not need to perform the actual simulation work like a real simulator does as described in Section 6.2.

We also implement a classic optimistic simulation protocol [Jef85] in AnTrace to evaluate the performance of the optimistic approach. In the current implementation, a state saving is performed for each channel read event (RX). In addition, a node always assumes no transmissions when it needs to guess because otherwise it has to guess the content of a transmission as well, which is difficult. No speedup techniques are implemented for the optimistic approach since we can model many of the techniques by adjusting the overhead times or controlling certain inputs to the playback process such as the message-withdraw-probability.

6.5 Evaluation

In this section, we conduct a series of experiments to evaluate SimVal and use SimVal as the tool to study the performance of both the conservative and optimistic approaches in simulating WSNs. For these experiments, simulation traces for different types of WSNs are collected first by simulating them with trace-enabled PolarLite on 2 of the 8 processors (Intel Xeon 2.0GHz) of a SMP server. The 2-CPU simulation traces are subsequently given to SimVal as inputs to estimate the performance of the conservative and optimistic approaches with various speedup techniques and on different numbers of CPUs.

Actual performance results for the conservative approach are also collected by simulating the same WSNs with non-trace-enabled PolarLite on the same server using different numbers of physical CPUs (cores). These actual measurements are used to validate the corresponding estimations from SimVal. The server runs Linux 2.6.31 with 8GBytes of RAM. Sun's Java 1.6.0 is used to execute both PolarLite and AnTrace. Simulation speed in units of MHz is calculated with Equation (6.1).

$$Speed = \frac{total\ number\ of\ simulated\ clock\ cycles/10^6}{(execution\ time) \times (number\ of\ nodes)} \tag{6.1}$$

The simulation overheads used to evaluate the conservative approach in the experiments (Table 6.2) are based on the average of those collected from real simulation

runs of non-trace-enabled PolarLite. Overheads beyond 8 CPUs cannot be collected directly since the server has only 8 CPUs. We instead collect the overheads using 1 to 7 CPUs and based on that linearly extrapolate the rest. The overheads for 8 CPUs are not collected since the process is likely to be interfered by other running processes. The overheads are presented in detail with the experiments.

For fair evaluations, the same overheads are also used to evaluate the optimistic approach when applicable. The overhead for sending or receiving an anti-message is set to be the same as sending or receiving a sync message since the two messages are very similar in size. We choose the state saving overhead and the rollback overhead to be the same as the context switch overhead because similar to a context switch, necessary states are saved or restored. For instance, a state saving would need to read the current state and save (write) it to memory. Note that in an actual optimistic simulation, memory occupied by safe state has to be released so new states can reuse them. We amortize those overheads into each state saving and rollback overhead.

If there are more nodes than the number of virtual CPUs in evaluating the optimistic approach, the round-robin scheduler is configured to simulate each node for $8000000ns$ of wallclock time before swapping in other nodes. This provides very good performance based on our experiments. Note that similar experiments have to be done in real optimistic simulations as well to find an interval that provides good performance. The message-withdraw-probability for non-transmitting nodes is set to be 0%. We also apply a 0% message-withdraw-probability to all the nodes throughout the experiment to provide reference points. For nodes that transmit packets, we experiment with message-withdraw-probabilities of 50% and 25%. This would give us a range of performance for cases where speedup techniques such as lazy cancellation are used to control such probabilities. We use the CountSend (sender) and CountReceive (receiver) programs from the TinyOS 1.1 distribution for our experiments. CountSend broadcasts the value of a continuously increasing counter at a fixed interval. CountReceive listens for transmissions from CountSend and displays the last received value on LEDs. These programs are similar to what other WSN simulators use in evaluating their performance [LLWC03, TLP05, WWM07]. Simulations are carried out by executing the programs on simulated Mica2 nodes [Cro08] and the starting time of each node is separated 10 clock cycles (ATMega128L microcontroller [Atm03]) apart. All simulations are run for 10 seconds of simulation time.

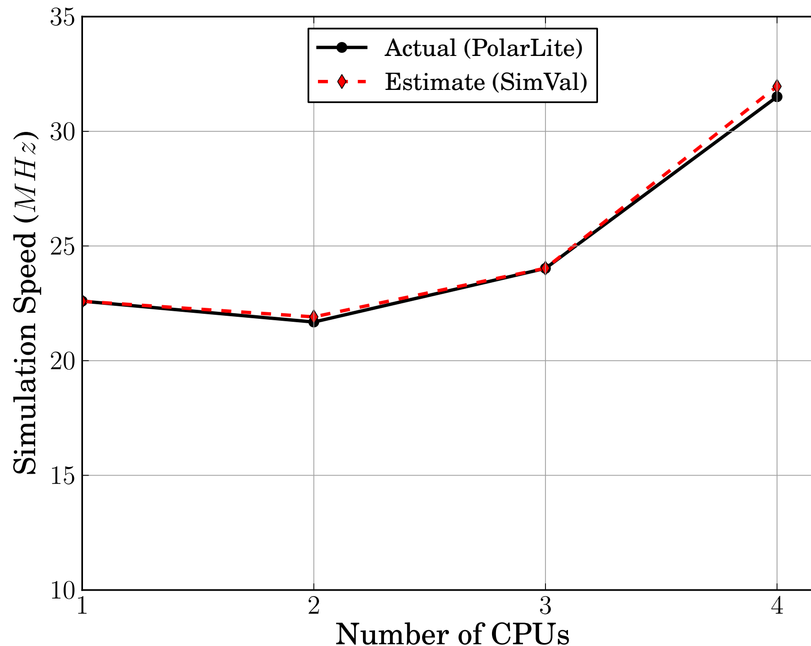

Figure 6.5: Actual and estimated simulation speed with the conservative approach on different numbers of CPUs (1 hop, 3 receivers and 1 sender)

6.5.1 Accuracy of SimVal

In this section, we conduct two sets of experiments to evaluate the accuracy of SimVal. In the first set of experiments, we simulate a WSN of 4 nodes. The 4 nodes consist of 1 sender and 3 receivers and all the nodes are within direct communication range of each other. The sender transmits a packet every $250ms$ and receivers are modeled as sink nodes that never transmit.

Figure 6.5 shows the actual measured speeds of simulating the WSN with non-trace-enabled PolarLite on 1 to 4 CPUs, as well as the estimated simulation speeds reported by SimVal using the conservative approach and the same numbers of virtual CPUs. As described earlier in this section, SimVal estimates the simulation speeds using a simulation trace collected from simulating the 4-node WSN with trace-enabled PolarLite on 2 CPUs. The overheads used for the estimation are shown in Figure 6.6. These overheads are based on actual measurements as described before and we notice that there is a significant increase of *context-switch* overhead from 1 to 2 CPUs. This is because when there is more than one CPU (core), threads may get moved across the CPUs during context switches. We also see in Figure 6.6 significant increases of *sync-*

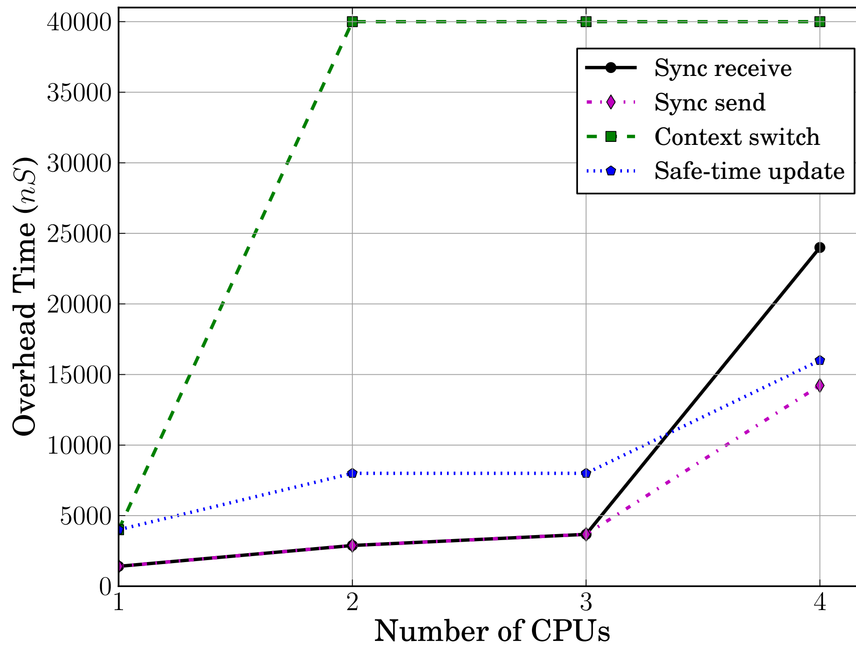

Figure 6.6: Simulation overheads on different numbers of CPUs (1 hop, 3 receivers and 1 sender)

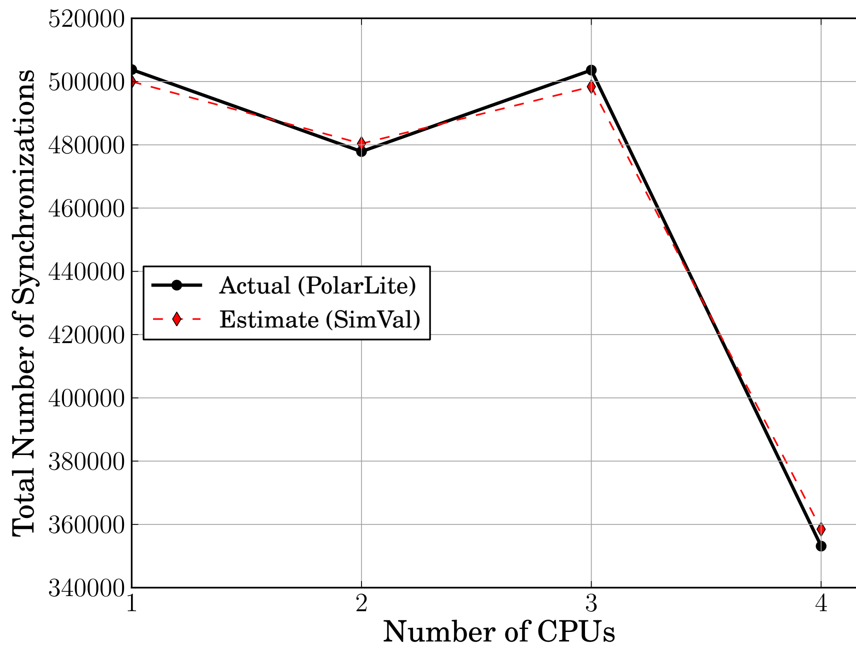

Figure 6.7: Actual and estimated total number of synchronizations with the conservative approach on different numbers of CPUs (1 hop, 3 receivers and 1 sender)

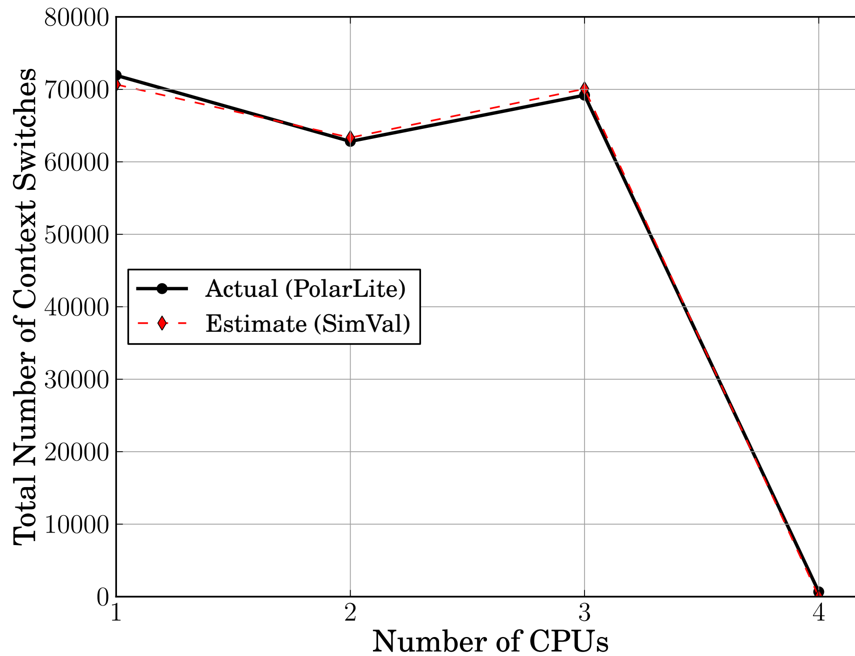

Figure 6.8: Actual and estimated total number of context switches with the conservative approach on different numbers of CPUs (1 hop, 3 receivers and 1 sender)

receive and *sync-send* overheads from 3 to 4 CPUs. This is due to the fact that sending and receiving messages in PolarLite requires atomic access to shared variables (memory). With 4 CPUs, all 4 nodes can run in parallel and the probability for them to wait for the shared accesses increases. Similarly, the *safe-time-update* overhead also increases significantly from 3 to 4 CPUs. Figure 6.7 and Figure 6.8 correspond to Figure 6.5 and show the actual and estimated total number of synchronizations and total number of context switches respectively. We can see that the estimated numbers match well with the actual ones.

The second set of experiments is similar to the first one except that we simulate a more sophisticated WSN with 12 senders and 12 receivers. The senders and receivers are configured to be the same as the ones in the first set of experiments and all nodes are within direct communication range of each other. The actual and estimated speeds of simulating this WSN with the conservative approach on different numbers of CPUs are shown in Figure 6.9. We can see that the estimated results match closely with the actual ones and the largest estimation error is around 5.8% under 7 CPUs. The actual simulation speeds in Figure 6.9 are collected by using up to 7 out of a total of 8 CPUs of

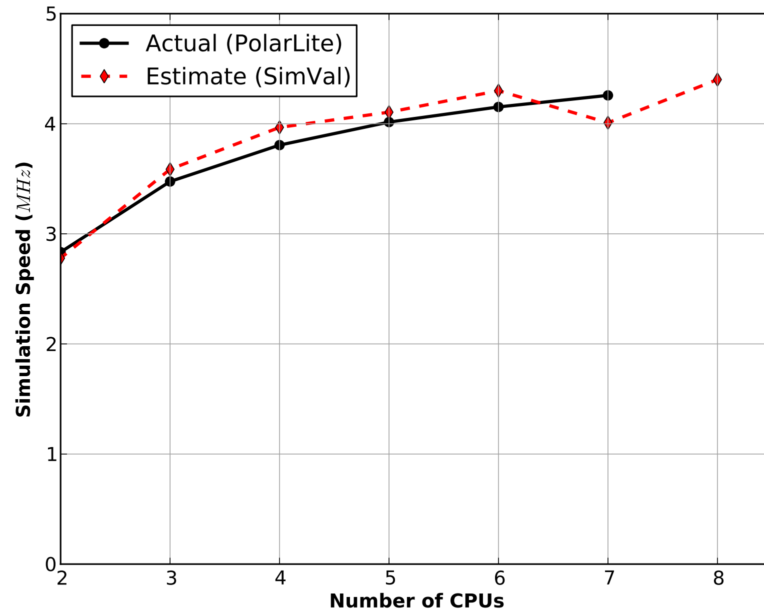

Figure 6.9: Actual and estimated simulation speed with the conservative approach on different numbers of CPUs (1 hop, 12 receivers and 12 senders)

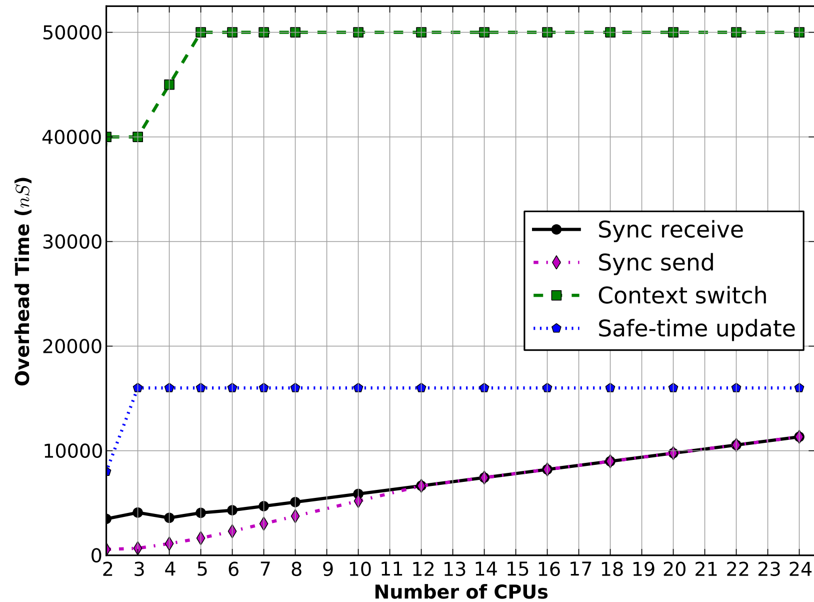

Figure 6.10: Simulation overheads on different numbers of CPUs (1 hop, 12 receivers and 12 senders)

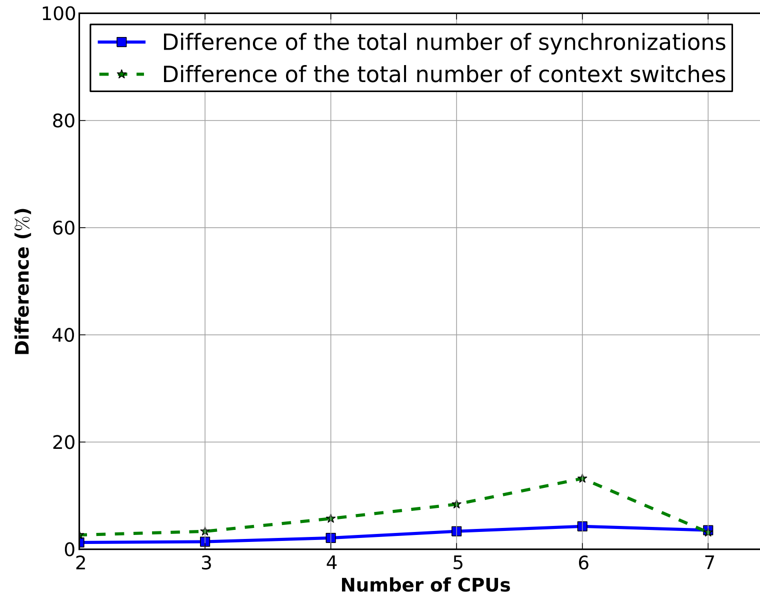

Figure 6.11: The percentage difference of the actual and estimated total number of synchronizations and context switches with the conservative approach on different numbers of CPUs (1 hop, 12 receivers and 12 senders)

the server. Not all the CPUs are used to avoid the interference of other running processes to the experiments. The overheads for simulating the WSN are shown in Figure 6.10. For this experiment, only those overheads for 2 to 7 CPUs are used and they are based on actual measurements. Compared to the overheads in the first set of experiments (Figure 6.6), we see an increase of the context switch overhead. This is due to the increased cost of thread scheduling which is performed via a shared semaphore (FIFO) in PolarLite. This is also caused by the increased overhead in moving threads across the boundary of the two physical processors (4 cores each) on the server when the number of the CPUs used in the experiments is more than 4.

Figure 6.11 corresponds to Figure 6.9 and shows the percentage differences of the actual and estimated total numbers of synchronizations and context switches with the conservative approach on different numbers of CPUs. We can see that with an error of less than 4.3%, the estimation for the total number of synchronizations is very accurate. The estimation for the total number of context switches has a maximum error of 13.2% under 6 CPUs. We believe this is because the total number of context switches in the experiments is more sensitive to the order the nodes are simulated when there are more nodes than the number of CPUs. In SimVal, the nodes are scheduled strictly following

the ascending order of the Node IDs but in real simulations the order can change due to concurrent events. For instance, two waiting nodes can be awoken at the same time by some different running nodes but only one of the awoken ones can be scheduled to run due to the lack of CPUs. In such a case, SimVal always schedules the node with a smaller ID first. However, in a real simulation, any one of the nodes can be selected to simulate first. It is important to note that simulation speed is not just affected by the number of context switches or synchronizations. It is also determined by how efficiently the CPUs are used for parallel simulations. In the worst case scenario, a node may become the bottleneck causing all other nodes to wait for it. As a result, only one CPU is actively used at that time.

6.5.2 Conservative vs. Optimistic in Simulating Single-hop WSNs

In this section, we use SimVal as a tool to study the performance of the conservative and optimistic approaches in simulating single-hop WSNs. The setup of our first experiment is the same as the last experiment in Section 6.5.1 (Figure 6.9). Basically, we evaluate the performance of simulating a WSN of 24 nodes that are within direct communication range of each other. The WSN consists of 12 senders and 12 receivers. Each sender transmits a packet every $250ms$ and receivers are modeled as sink nodes that never transmit.

Figure 6.12 shows the estimated speeds, reported by SimVal, in simulating the WSN with both the conservative and optimistic approaches on 2 to 24 CPUs. The actual measured simulation speeds with the conservative approach using 2 to 7 CPUs are also shown in Figure 6.12 as references. As shown in Figure 6.12, the optimistic approach is significantly more scalable over the number of CPUs than the conservative one. In other words, the optimistic approach can better take advantage of the increased parallelism in the simulation when the number of CPUs increases. The parallelism in a distributed simulation increases with the number of CPUs because more nodes can be simulated in parallel on the CPUs, as long as the number of CPUs is less than the number of nodes. However, the conservative approach, especially with the LazySync speedup technique, provides faster simulation speed when the number of CPUs is small. This suggests if there are not enough number of CPUs to simulate nodes in parallel, or in other words, the ratio of the number of nodes over the number of CPUs is high, the conservative approach can provide a better performance in terms of simulation speed. The overheads used by

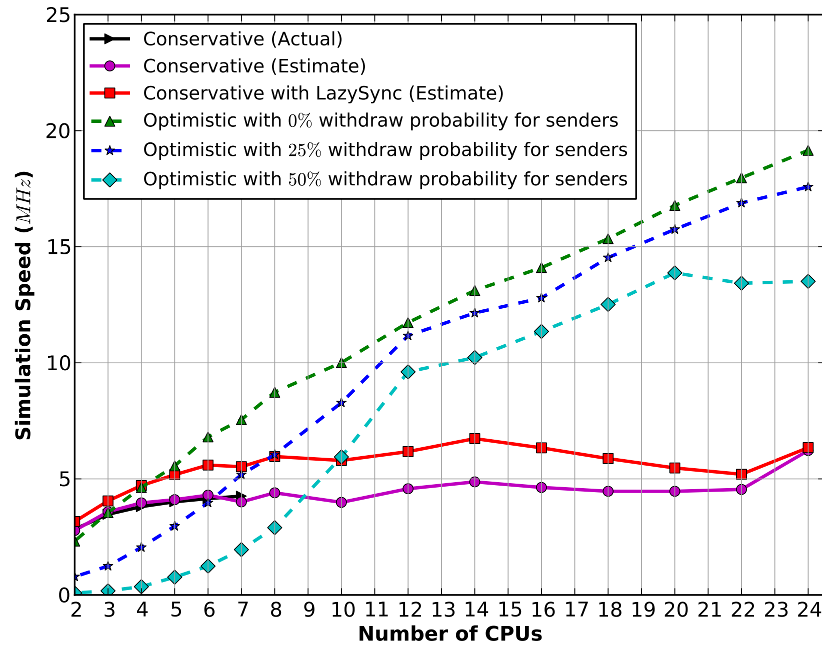

Figure 6.12: Estimated simulation speeds with the conservative and optimistic approaches (1 hop, 12 receivers and 12 senders)

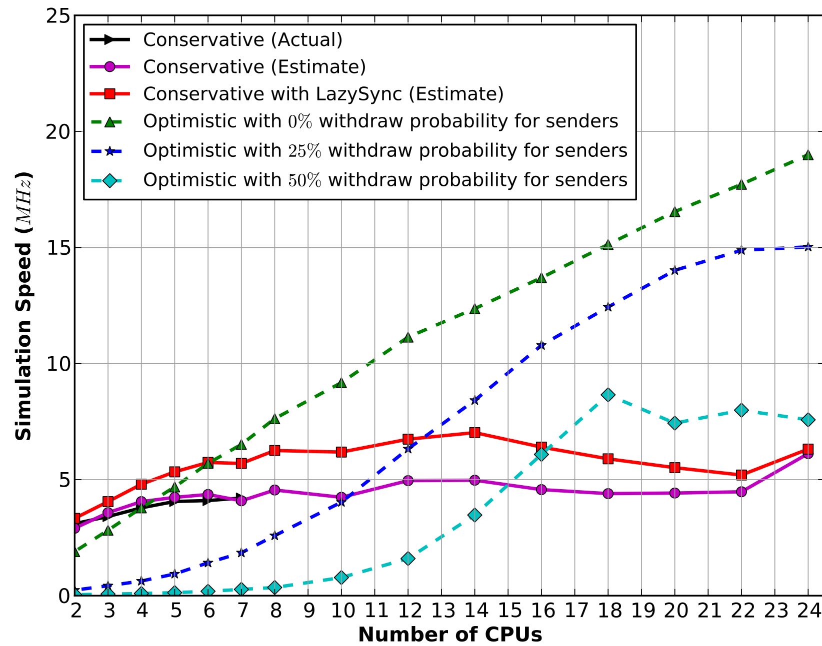

Figure 6.13: Estimated simulation speeds with the conservative and optimistic approaches (1 hop, 6 receivers and 18 senders)

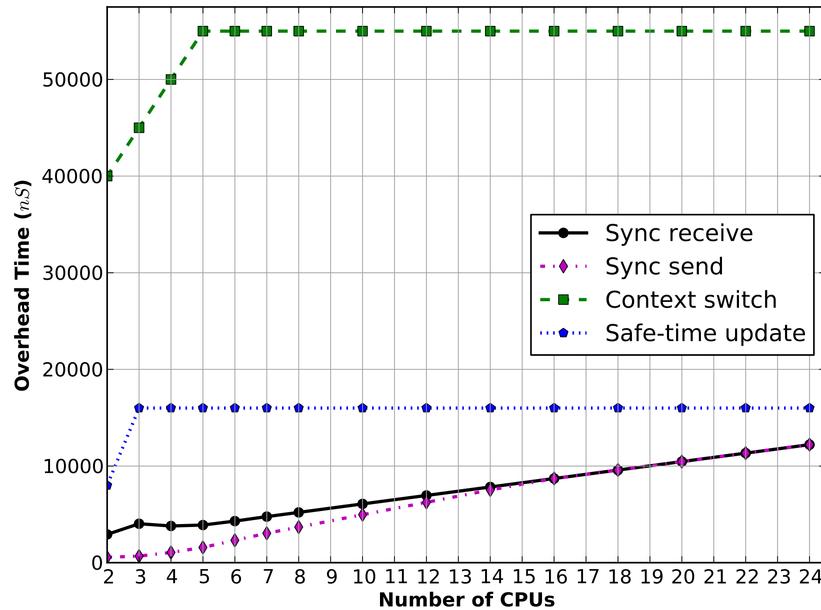

Figure 6.14: Simulation overheads on different number of CPUs (1 hop, 6 receivers and 18 senders)

SimVal for these evaluations are shown in Figure 6.10. Among them, the overheads for 8 or more CPUs are linearly extrapolated from the overheads of 5 to 7 CPUs. To keep the overheads consistent, we also ensure that the *sync-send* overhead never exceeds the *sync-receive* after the linear extrapolations.

Our second set of experiments is similar to the first one in this section except that we increase the number of senders in the WSN to 18 and reduce the number of receivers to 6. This allows us to study the performance of the conservative and optimistic approaches in a busier WSN, a network that has more transmissions. The estimated simulation speeds for this WSN are shown in Figure 6.13. Compared to Figure 6.12, we can see that the optimistic approach has worse performance in a busier network but can still outperform the conservative approach if the ratio of the number of nodes over the number of CPUs is high enough to allow sufficient parallelism in the simulation. On the other hand, the performance of the conservative approach is not affected by the increased transmissions. The overheads used for evaluating the WSN are shown in Figure 6.14. They are comparable to the ones in Figure 6.10.

6.5.3 Conservative vs. Optimistic in Simulating Multi-hop WSNs

In the multi-hop experiments, we use SimVal to study the performance of the conservative and optimistic approaches in simulating a WSN service that floods data to every node in a multi-hop WSN. This service works by having every node in the WSN relay, by broadcasting, messages it receives. To avoid sending duplicate messages, a relay node only forwards messages with IDs greater than the largest IDs of the messages it has already sent.

For experiments in this section, we modify CountReceive to relay messages the way we just described. The WSN we simulate has 25 nodes laid 10 meters apart on a 5 by 5 grid. A corner node is configured as the sender and the rest of nodes are configured as relaying nodes (forwarders) running the modified CountReceive program. The sender transmits a new packet every $500ms$ with an increasing ID. All radios are duty cycled by setting the B-MAC radio-level duty cycling mode [PHC04] to 4 ($185ms$). The transmission range of all nodes is set to 19 meters.

Based on the estimated simulation speeds from SimVal, Figure 6.15 compares the performance of the conservative and optimistic approaches in simulating the WSN with 2 to 24 CPUs. We can see that in the multi-hop scenario, the optimistic approach is also more scalable over the number of CPUs than the conservative one. Unlike the cases in the previous single-hop experiments, the performance of the optimistic approach is very close to the conservative one when the number of CPUs is small. This suggests that the optimistic approach can naturally take advantage of the duty cycling latency in the network. However, with *LazySync*, the conservative approach also scales well with the number of CPUs in this experiment and outperforms the optimistic approach when the number of CPUs is less than 16. In addition, the conservative approach can also benefit from the delays using the speedup techniques proposed in [JG09a]. Figure 6.15 also shows that the message-withdraw-probability has little effect toward the performance of the optimistic approach in this experiment. This is because compared to the previous single-hop experiments, there are a smaller number of transmissions in this WSN due to increased transmission interval and duty cycling. This leads to a smaller number of rollbacks in the optimistic simulations. The overheads used for this experiment are comparable to the ones in Figure 6.10.

To study the performance of the conservative and optimistic approaches over a quieter multi-hop network, we increase the sender's transmission interval in the pre-

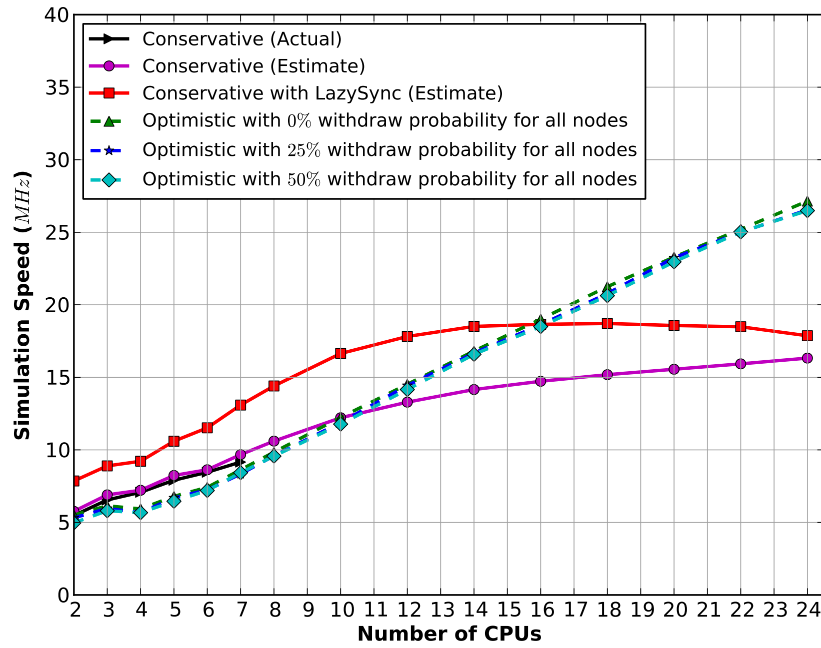

Figure 6.15: Estimated simulation speeds with the conservative and optimistic approaches (multi-hop, 24 forwarders and 1 sender, 500ms transmission interval)

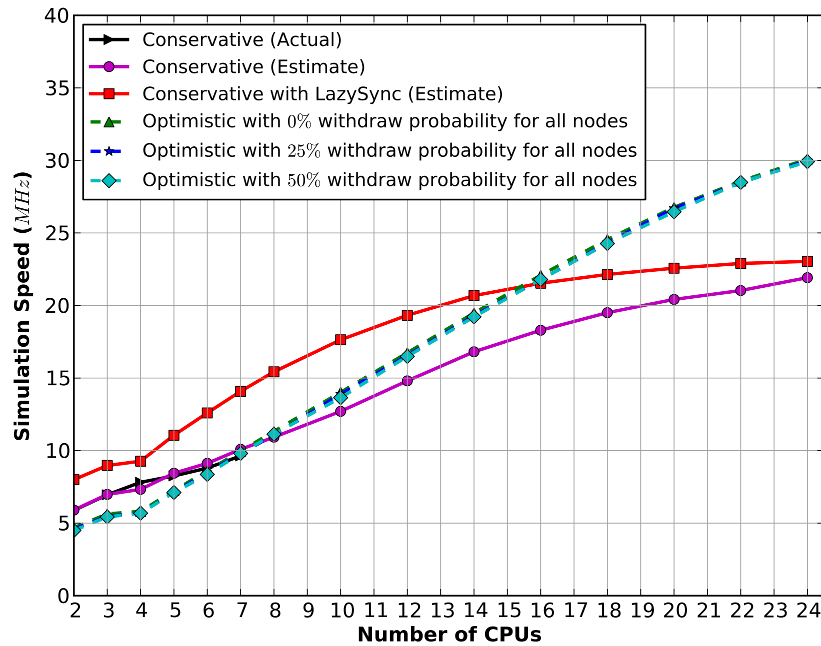

Figure 6.16: Estimated simulation speeds with the conservative and optimistic approaches (multi-hop, 24 forwarders and 1 sender, 1000ms transmission interval)

vious experiment to 1000ms and the results are shown in Figure 6.16. Compared to Figure 6.15, we can see that the increase of transmission interval further improves the relative performance of the optimistic approach and the improvements are consistent over the entire CPU range. We can also see that the performance of the conservative approach improves as well and with *LazySync* the conservative approach still outperforms the optimistic one when the number of CPUs is less than 16. The overheads used for this experiment are also comparable to the ones in Figure 6.10.

6.6 Related Work

The trace based evaluation technique of SimVal is similar in concept to trace-driven simulations [UM97]. Trace-driven simulations are commonly used to study the performance of various algorithms or techniques based on traces collected from real systems. For example, a trace-driven memory simulation collects memory access traces from real computers and uses that to evaluate the performance of different cache protocols. Similarly, with SimVal, traces of various simulation events are collected from actual simulations and used to evaluate the performance of different simulation approaches and algorithms.

There is a large body of research on evaluating the performance of the conservative and optimistic approaches in simulating many different types of applications. They are either based on direct comparisons [QB95, Fuj99b] or rely heavily on analytic methods [ACD+92, Nic93, TTA98, Son01]. For example, the experiments in [QB95] directly compare the performance of the conservative and optimistic approaches on simulating the memory hierarchies of some multiprocessor systems using 12 CPUs and show that the performance of the optimistic approach is slightly better. The author in [Nic93] analyzes the performance of the conservative approach based on a stochastic model and shows that a great deal of work can be done in parallel if there is a great deal of concurrent activity in the simulation models. Analytical approaches generally require a deep understanding of the application under simulation so important properties can be extracted to construct the appropriate analytical models. However, this is not easy in the case of simulating WSNs due to the complexity of the simulation models, the intricate interactions of a large number of sensor nodes, and the fact that there exist significant amounts of diversity in WSNs and their applications. In addition, the performance of the optimistic approach is very difficult to analyze and even most advanced analytical

models are limited to certain simulation schemes [TTA98].

6.7 Summary

In this chapter, we have presented a technique to evaluate the performance of the conservative and optimistic simulation approaches based on simulation traces. The technique is independent of the trace collection process and separates the simulation overheads from the actual simulation algorithms. This makes it possible to use the same traces to quickly prototype and study the performance of both approaches with any design tradeoffs, speedup techniques and optimizations. In addition, this technique can be used to evaluate simulation performance on any simulation platforms or any number of CPUs without using real hardware.

The trace-based evaluation technique is developed in the SimVal framework and the trace collection process is implemented as a part of the PolarLite simulator. Using these tools, we conducted extensive experiments to evaluate the technique and study the performance of the conservative and optimistic approaches in simulating WSNs. The experimental results show that the technique is accurate and the optimistic approach is more scalable over the number of CPUs than the conservative one in simulating WSNs. The results also show that the conservative approach has better performance if the ratio of the number of nodes to the number of CPUs is large. This suggests that a unified approach that supports both the conservative and optimistic approaches at the same time during a simulation is most suitable for simulating WSNs. Protocols such as the one in [JB94] have been proposed for exactly such purposes. SimVal is an ideal tool to quickly prototype and study such protocols for simulating WSNs before the actual implementation of a new unified WSN simulator.

Acknowledgements: Chapter 6, in part, has been submitted for publications as “A Framework for Evaluating the Performance of Conservative and Optimistic Approaches in Simulating Sensor Networks” by Zhong-Yi Jin and Rajesh Gupta. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Conclusions and Future Work

To meet the large computational requirements for simulating wireless sensor networks with high fidelity, wireless sensor network simulators employ distributed simulation techniques to leverage the combined resources of multiple processors or computers. This technique is becoming increasingly popular due to the emergence of low cost multi-core processors and the wide availability of cloud-computing infrastructures. Distributed sensor network simulators are essentially sequential discrete event simulators running in parallel on multiple processing elements. The fundamental challenge in these multi-simulation frameworks is how individual simulators coordinate and control temporal advances in the simulation models on local simulations. Errors in this process may lead to causality violations which occur when a simulator temporally gets too far ahead in evaluating local events before the causative events from other simulators are incorporated into the simulation.

Distributed sensor network simulators can be designed based on two different approaches: conservative and optimistic. These two approaches are fundamentally different in how they preserve causality relations in simulations. The conservative approach seeks to achieve this by advancing local simulation time to the extent that the simulation is provably correct and guaranteed to be free of causality violations. On the other hand, the optimistic approach is more aggressive and may actually advance local time without taking into account all causality constraints. To recover from causality violations, optimistic simulators feature mechanisms such as anti-messages to roll back simulations as needed.

The large overheads in preserving causality during distributed simulations of

WSNs result in a significant increase in simulation time. Given that existing WSN simulators are based on the conservative approach, in this dissertation, we have developed and generalized three lookahead-time-based techniques that improve the performance of the conservative approach in simulating WSNs by minimizing the number of sensor node synchronizations and increasing parallelism during simulations. We have also developed LazySync, a synchronization scheme that further improves the performance of the conservative approach by identifying and eliminating unnecessary synchronizations during simulations. Using these techniques, we have developed the PolarLite simulator [JG08] which is a fully functional cycle accurate distributed simulation framework built on top of the Avrora simulator [TLP05] from UCLA. Based on extensive experiments, we have demonstrated that PolarLite provides better simulation performance in terms of simulation speed and scalability than any other existing WSN simulators of similar accuracy.

Since events are handled fundamentally differently across conservative and optimistic simulators and there exist a large number of design tradeoffs, potential speedup techniques and optimizations for each of the approaches, it is difficult to choose an approach for a specific sensor network and its applications. In addition, a simulator is usually implemented and optimized for a specific hardware architecture such as symmetric multiprocessors (SMPs) or clusters, making it challenging to compare simulation performance over different hardware architectures. Finally, a direct performance comparison is limited to the specific computing platforms or the number of CPUs that are available. In this dissertation, we have presented SimVal, a framework that evaluates the performance of the approaches accurately using simulation traces. SimVal separates simulation overheads from actual simulation algorithms and allows one to use the same traces to evaluate both approaches with any design tradeoffs, speedup techniques and optimizations on virtual computers with any number of CPUs. We have implemented SimVal and use it to compare the performance of the two approaches in simulating WSNs. Our evaluations show the conservative approach has better performance when the ratio of the number of nodes over the number of simulation CPUs is high and the gain is especially significant in simulating busy networks. The optimistic approach is much more scalable over a large number of CPUs and performs better in quieter networks.

Based on the study of the relative performance of the conservative and optimistic approaches in this dissertation, we have shown that a unified simulator that

supports both of the approaches at the same time during a simulation is most suitable for simulating WSNs. There are many interesting problems that associate with using the unified approach, for example, how nodes should be partitioned so different simulation approaches can be used on different nodes. SimVal is an ideal tool for quickly prototyping and studying the solutions of such problems before the actual implementation of a new unified WSN simulator.

Another interesting direction to explore as future work is to automatically identify the performance bottleneck in a simulation and enable appropriate techniques to address the problems. This could be a natural extension of SimVal.

In this dissertation, SimVal is used to evaluate the performance of WSN simulators but the basic trace-based evaluation technique is general. It can be used to evaluate simulation performance in other application areas as well. For example, it can be used to evaluate the performance of SystemC simulations for system-level modeling [SSG05]. In fact, since an ideal-trace completely separates domain specific knowledge about how to model and simulate a specific application from the basic simulation algorithms, SimVal can potentially be extended to evaluate the performance of any kinds of simulations. We also plan to support in SimVal the capability to evaluate simulation performance on heterogeneous CPUs of different speeds. This can be achieved by scaling the real-work-times in the playback process.

Bibliography

[ACD+92] Ian F. Akyildiz, Liang Chen, Samir R. Das, Richard M. Fujimoto, and Richard F. Serfozo. Performance analysis of "time warp" with limited memory. *SIGMETRICS Perform. Eval. Rev.*, 20(1):213–224, 1992.

[Atm03] Atmel. *ATMega128L Datasheet*, 2003.

[BCD+05] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, 2005.

[CBMPS04] G. Chen, J. Branch, L. Zhu M. Pflug, and B. Szymanski. *SENSE: A Wireless Sensor Network Simulator*. Advances in Pervasive Computing and Networking. Springer, New York, NY, 2004.

[Cha99] Xinjie Chang. Network simulations with opnet. In *Simulation Conference Proceedings, 1999. Winter*, volume 1, pages 307–314, 5-8 Dec. 1999.

[CLZ06] Elaine Cheong, Edward A. Lee, and Yang Zhao. Viptos: A graphical development and simulation environment for tinyos-based wireless sensor networks. Technical Report UCB/EECS-2006-15, EECS Department, University of California, Berkeley, Feb 2006.

[CM79] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, SE-5(5):440–452, Sept. 1979.

[CM81] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206, 1981.

[Cro08] Crossbow. *MICA2 Datasheet*, 2008.

[Den82] Monty M. Denneau. The yorktown simulation engine. In *DAC '82: Proceedings of the 19th Design Automation Conference*, pages 55–59, Piscataway, NJ, USA, 1982. IEEE Press.

[FKM92] David Filo, David C. Ku, and Giovanni De Micheli. Optimizing the control-unit through the resynchronization of operations. *Integr. VLSI J.*, 13(3):231–258, 1992.

[Fuj99a] Richard M. Fujimoto. Parallel and distributed simulation. In *WSC '99: Proceedings of the 31st conference on Winter simulation*, pages 122–131, New York, NY, USA, 1999. ACM.

[Fuj99b] Richard M. Fujimoto. *Parallel and Distribution Simulation Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[Gaf88] A. Gafni. Rollback mechanisms for optimistic distributed simulation systems. In *SCS Multiconference on Distributed Simulation*, volume 19, pages 61–67, July 1988.

[GEC+04] Lewis Girod, Jeremy Elson, Alberto Cerpa, Thanos Stathopoulos, Nithya Ramanathan, and Deborah Estrin. Emstar: a software environment for developing and deploying wireless sensor networks. In *Proceedings of the 2004 USENIX Technical Conference*, Boston, MA, 2004. USENIX.

[GLvB+03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[GSR+04] Lewis Girod, Thanos Stathopoulos, Nithya Ramanathan, Jeremy Elson, Deborah Estrin, Eric Osterweil, and Tom Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems*, Baltimore, MD, 2004. ACM.

[HC02] Jason L. Hill and David E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, 2002.

[Hen08] Tom Henderson. *NS-3 Overview*, 2008.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, 2000.

[Hug89] J. Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989.

[JB94] Vikas Jha and Rajive L. Bagrodia. A unified framework for conservative and optimistic distributed simulation. *SIGSIM Simul. Dig.*, 24(1):12–19, 1994.

[JBW+87] D. Jefferson, B. Beckman, F. Wieland, L. Blume, and M. Diloreto. Time warp operating system. *SIGOPS Oper. Syst. Rev.*, 21(5):77–93, 1987.

[Jef85] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[JG08] ZhongYi Jin and Rajesh Gupta. Improved distributed simulation of sensor networks based on sensor node sleep time. In *DCOSS '08: Proceedings of the 4th IEEE International Conference on Distributed Computing in Sensor Systems*, pages 204–218, Santorini Island, Greece, 2008.

[JG09a] ZhongYi Jin and Rajesh Gupta. Improving the speed and scalability of distributed simulations of sensor networks. In *IPSN '09: Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 169–180, San Francisco, California, USA, 2009.

[JG09b] ZhongYi Jin and Rajesh Gupta. Lazysync: A new synchronization scheme for distributed simulation of sensor networks. In *DCOSS '09: Proceedings of the 5th IEEE International Conference on Distributed Computing in Sensor Systems*, pages 103–116, Marina del Rey, California, USA, 2009.

[JG09c] ZhongYi Jin and Rajesh Gupta. Rssi based location-aware pc power management. In *HotPower 09*, 2009.

[JSG07] ZhongYi Jin, Curt Schurgers, and Rajesh Gupta. An embedded platform with duty-cycled radio and processing subsystems for wireless sensor networks. In *International Workshop on Systems, Architectures, Modeling, and Simulation (SAMOS)*, 2007.

[JSG09] ZhongYi Jin, Curt Schurgers, and Rajesh K. Gupta. A gateway node with duty-cycled radio and processing subsystems for wireless sensor networks. *ACM Trans. Design Autom. Electr. Syst.*, 14(1), 2009.

[KP82] E. Kronstadt and G. Pfister. Software support for the yorktown simulation engine. In *DAC '82: Proceedings of the 19th Design Automation Conference*, pages 60–64, Piscataway, NJ, USA, 1982. IEEE Press.

[LAW08] Olaf Landsiedel, Hamad Alizai, and Klaus Wehrle. When timing matters: Enabling time accurate and scalable simulation of sensor network applications. In *IPSN '08: Proceedings of the 2008 International Conference on Information Processing in Sensor Networks*, pages 344–355, Washington, DC, USA, 2008. IEEE Computer Society.

[Lev06] Philip Levis. *TinyOS Programming*, June 2006.

[LGH+05] P. Levis, D. Gay, V. Handziski, J.-H.Hauer, B.Greenstein, M.Turon, J.Hui, K.Klues, C.Sharp, R.Szewczyk, J.Polastre, P.Buonadonna, L.Nachman, G.Tolle, D.Culler, and A.Wolisz. T2: A second generation os for embedded sensor networks. Technical Report TKN-05-007, Telecommunication Networks Group, Technische Universität Berlin, November 2005.

[LLWC03] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM Press.

[LN02] Jason Liu and David M. Nicol. Lookahead revisited in wireless network simulations. In *PADS '02: Proceedings of the sixteenth workshop on Parallel and distributed simulation*, pages 79–88, Washington, DC, USA, 2002. IEEE Computer Society.

[MSK+05] C. Mallanda, A. Suri, V. Kunchakarra, S.S. Iyengar, R. Kannan, and A. Durresi. Simulating wireless sensor networks with omnet++. Sensor Network Research Group, Department of Computer Science, Louisiana State University, Baton Rouge, LA., 2005.

[Nes] *NesCT*.

[Nic93] David M. Nicol. The cost of conservative synchronization in parallel discrete event simulations. *J. ACM*, 40(2):304–333, 1993.

[Nic98] David M. Nicol. Scalability, locality, partitioning and synchronization pdes. *SIGSIM Simul. Dig.*, 28(1):5–11, 1998.

[NS2] *The ns-2 Manual*.

[OAVRHR05] E. Ould-Ahmed-Vall, G.F. Riley, B.S. Heck, and D. Reddy. Simulation of large-scale sensor networks using gtsnets. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*, pages 211–218, 27-29 Sept. 2005.

[ODE+06] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with cooja. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*, pages 641–648, Nov. 2006.

[PBM+04] J. Polley, D. Blazakis, J. McGee, D. Rusk, and J.S. Baras. Atemu: a fine-grained sensor network simulator. In *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, pages 145–152, 4-7 Oct. 2004.

[Pfi86] G.F. Pfister. The ibm yorktown simulation engine. *Proceedings of the IEEE*, 74(6):850 – 860, june 1986.

[PHC04] Joseph Polastre, Jason Hill, and David Culler. Versatile low power media access for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 95–107, New York, NY, USA, 2004. ACM.

[PN02] L.F. Perrone and D.M. Nicol. A scalable simulator for tinyos applications. In *Simulation Conference, 2002. Proceedings of the Winter*, volume 1, pages 679–687, 8-11 Dec. 2002.

[PSC05] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: Enabling ultra-low power wireless research. In *the Fourth International Conference on Information Processing in Sensor Networks*, 2005.

[PSS00] Sung Park, Andreas Savvides, and Mani B. Srivastava. Sensorsim: a simulation framework for sensor networks. In *MSWIM '00: Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 104–111, New York, NY, USA, 2000. ACM Press.

[PSS01] Sung Park, A. Savvides, and M.B. Srivastava. Simulating networks of wireless sensors. In *Simulation Conference, 2001. Proceedings of the Winter*, volume 2, pages 1330–1338, 9-12 Dec. 2001.

[QB95] Xiaohan Qin and J.-L. Baer. A comparative study of conservative and optimistic trace-driven simulations. In *SS '95: Proceedings of the 28th Annual Simulation Symposium*, page 42, Washington, DC, USA, 1995. IEEE Computer Society.

[RAF+04] George F. Riley, Mostafa H. Ammar, Richard M. Fujimoto, Alfred Park, Kalyan Perumalla, and Donghua Xu. A federated approach to distributed network simulation. *ACM Trans. Model. Comput. Simul.*, 14(2):116–148, 2004.

[SCH+05] Ahmed Sobeih, Wei-Peng Chen, Jennifer C. Hou, Lu-Chuan Kung, Ning Li, Hyuk Lim, Hung-Ying Tyan, and Honghai Zhang. J-sim: A simulation environment for wireless sensor networks. In *ANSS '05: Proceedings of the 38th annual Symposium on Simulation*, pages 175–187, Washington, DC, USA, 2005. IEEE Computer Society.

[SCH+06] Ahmed Sobeih, Wei-Peng Chen, Jennifer C. Hou, Lu-Chuan Kung, Ning Li, Hyuk Lim, Hung-Ying Tyan, and Honghai Zhang. J-sim: A simulation and emulation environment for wireless sensor networks. *Wireless Communications, IEEE [see also IEEE Personal Communications]*, 13:104–119, 2006.

[SG08] Ryo Sugihara and Rajesh K. Gupta. Programming models for sensor networks: A survey. *ACM Trans. Sen. Netw.*, 4(2):1–29, 2008.

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

[SMB00] Ha Yoon Song, Richard A. Meyer, and Rajive Bagrodia. An empirical study of conservative scheduling. In *PADS '00: Proceedings of the fourteenth workshop on Parallel and distributed simulation*, pages 165–172, Washington, DC, USA, 2000. IEEE Computer Society.

[SML+04] Gyula Simon, Miklós Maróti, Ákos Lédeczi, György Balogh, Branislav Kusy, András Nádas, Gábor Pap, János Sallai, and Ken Frampton. Sensor network-based countersniper system. In *SenSys '04: Proceedings of*

the 2nd international conference on Embedded networked sensor systems, pages 1–12, New York, NY, USA, 2004. ACM.

[SMP+04] Robert Szewczyk, Alan Mainwaring, Joseph Polastre, John Anderson, and David Culler. An analysis of a large scale habitat monitoring application. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 214–226, New York, NY, USA, 2004. ACM Press.

[Son01] Ha Yoon Song. A probabilistic performance model for conservative simulation protocol. In *PADS '01: Proceedings of the fifteenth workshop on Parallel and distributed simulation*, pages 200–207, Washington, DC, USA, 2001. IEEE Computer Society.

[SOP+04] Robert Szewczyk, Eric Osterweil, Joseph Polastre, Michael Hamilton, Alan Mainwaring, and Deborah Estrin. Habitat monitoring with sensor networks. *Commun. ACM*, 47(6):34–40, 2004.

[SPMC04] Robert Szewczyk, Joseph Polastre, Alan M. Mainwaring, and David E. Culler. Lessons from a sensor network expedition. In *EWSN*, pages 307–322, 2004.

[SSG05] N. Savoiu, Sandeep Shukla, and Rajesh Gupta. Improving systemc simulation through petri net reductions. In *MEMOCODE '05*, pages 131–140, Washington, DC, USA, 2005. IEEE Computer Society.

[TA] TinyOS-Alliance. Tinyos 1.1.15.

[TLP05] Ben L. Titzer, Daniel K. Lee, and Jens Palsberg. Avrora: scalable sensor network simulation with precise timing. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, pages 477–482, Piscataway, NJ, USA, 2005. IEEE Press.

[TTA98] Seng Chuan Tay, Yong Meng Teo, and Rassul Ayani. Performance analysis of time warp simulation with cascading rollbacks. *SIGSIM Simul. Dig.*, 28(1):30–37, 1998.

[UM97] Richard A. Uhlig and Trevor N. Mudge. Trace-driven memory simulation: a survey. *ACM Comput. Surv.*, 29(2):128–170, 1997.

[Var01] A. Varga. The omnet++ discrete event simulation system. In *European Simulation Multiconference*, Prague, Czech Republic, June 2001.

[VXSB07] Maneesh Varshney, Defeng Xu, Mani Srivastava, and Rajive Bagrodia. Senq: a scalable simulation and emulation environment for sensor networks. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 196–205, New York, NY, USA, 2007. ACM Press.

[WALJ+06] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. Fidelity and yield in a volcano monitoring sensor network. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 27–27, Berkeley, CA, USA, 2006. USENIX Association.

[WGC+06] Ye Wen, S. Gurun, N. Chohan, R. Wolski, and C. Krintz. Simgate: Full-system, cycle-close simulation of the stargate sensor network intermediate node. In *Embedded Computer Systems: Architectures, Modeling and Simulation, 2006 International Conference on*, pages 129–136, July 2006.

[Wor05] Tinyos 2.0. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 320–320, New York, NY, USA, 2005. ACM.

[WS06] Georg Wittenburg and Jochen Schiller. Running real-world software on simulated wireless sensor nodes. In *Proceedings of the ACM Workshop on Real-World Wireless Sensor Networks (REALWSN'06)*, pages 7–11, Uppsala, Sweden, June 2006.

[WWM07] Ye Wen, Rich Wolski, and Gregory Moore. Disens: scalable distributed sensor network simulation. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 24–34, New York, NY, USA, 2007. ACM Press.

[YHE02] Wei Ye, John Heidemann, and Deborah Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Infocom '02,*, pages 1567–1576, New York, NY, 2002.