Diss. ETH No. 17189

Distributed Embedded Systems
—
Validation Strategies

A dissertation submitted to the

SWISS FEDERAL INSTITUTE OF TECHNOLOGY
ZURICH

for the degree of

Doctor of Sciences

presented by

MATTHIAS DYER
Dipl. El.-Ing. ETH Zurich

born 16.10.1976
citizen of Winterthur, ZH

accepted on the recommendation of

Prof. Dr. Lothar Thiele, examiner
Prof. Dr. Koen Langendoen, co-examiner

2007

Seite Leer /
Blank leaf

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

TIK-SCHRIFTENREIHE NR. 89

MATTHIAS DYER

Distributed Embedded Systems

—

Validation Strategies

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Abstract

Validation plays an important role in the design process of embedded systems. It is the only way to relate the requirements and specification of a design to the real, final system. It is further a mean to obtain objective quality metrics about a system in its target environment, and to gain confidence in the systems quality. The validation of embedded systems is challenging. For complex industrial products, about 50 percent of the complete design time is spent on validation. The validation of distributed embedded systems is even more complex since these systems introduce new challenges such as (i) unreliable wireless communication, (ii) the distributed nature of the system, (iii) limited resources, and (iv) limited infrastructure for inspecting the system.

In this thesis we contribute towards an increased quality of the design of distributed embedded systems. Our research on validation strategies addresses the challenging problems that arise from the peculiarities of distributed embedded systems such as wireless sensor networks, reconfigurable-, and wearable computers. The specific contributions are presented and discussed for different phases and levels of abstraction in the design and development process:

- An estimation-based validation strategy is presented for wearable systems consisting of distributed modules with computing- and communication devices.

- We propose the virtualized execution in distributed embedded systems when applications are executed on reconfigurable hardware. We introduce hardware tasks and an interpreted coordination language which allows a developer to validate and to deploy tasks separately.

- We discuss the practicability of distributed algorithms and problematic simulation assumptions on the example of the topology control problem for wireless sensor networks.

- The Deployment Support Network (DSN) is proposed as a novel validation strategy for distributed wireless embedded systems. The DSN is a platform-independent toolkit that allows a developer to test applications on the real hardware and in a real-world deployment.

Zusammenfassung

Die Validierung spielt eine wichtige Rolle beim Entwurf von Eingebetteten Systemen. Es ist der einzige Weg um die Anforderungen und Spezifikationen eines Entwurfes mit dem entgültigen System in Beziehung zu bringen. Es ist weiter ein Mittel um ein sachliches Qualitätsmass eines Systemes zu erhalten. Die Validierung von Eingebetteten Systemen ist anspruchsvoll. Für komplexe industrielle Produkte wird ca. 50 Prozent der gesammten Entwicklungszeit mit Validierung verbracht. Die Validierung von Verteilten Eingebetteten Systemen ist sogar noch schwieriger, weil diese Systeme neue Schwierigkeiten mit sich bringen, wie z.B. (i) die unzuverlässige Funkkommunikation, (ii) die verteilte Eigenschaft des Systems, (iii) beschränkten Betriebsmittel und (iv) beschränkte Infrastruktur für die Beobachtung des Systems.

In dieser Arbeit tragen wir zu einer verbesserten Qualität im Entwurf von Verteilten Eingebetteten Systemen bei. Unsere Forschung über Validierungs Strategien behandelt anspruchsvolle Probleme welche von den Eigenheiten von Verteilten Eingebetteten Systemen wie drahtlose Sensornetze, Reconfigurable- und Wearable Computer herrühren. Wir stellen die einzelnen Beiträge für verschiedene Abschnitte und Abstraktionsstufen im Entwurf- und Entwicklungsprozess vor:

- Es wird ein Schätzungsverfahren als Validierungs Strategie für Wearable Systems präsentiert, die aus verteilten Modulen mit Rechner- und Kommunikationseinheiten bestehen.
- Wir schlagen eine virtualisierte Ausführung in Verteilten Eingebetteten Systemen vor, falls Anwendungen auf rekonfiguierbarer Hardware ausgeführt wird. Wir stellen Hardware Tasks und eine interpretierte Koordinierungssprache vor, was eine separate Validierung und Verteilung von Tasks ermöglicht.
- Wir disskutieren die Anwendbarkeit von verteilten Algorithmen und problematischen Simulationsannahmen am Beispiel des Topology Control Problems für Drahtlose Sensornetze.
- Das Deployment Support Network (DSN) wird vorgestellt als eine neuartige Validierungsstrategie für Verteilte Eingebettete Funksysteme. Das DSN ist ein plattformunabhängiges Werkzeugset welches dem Entwickler das Testen von Anwendungen auf der richtigen Hardware und im echten Betrieb ermöglicht.

Acknowledgement

First of all I would like to express my sincere gratitude to Prof. Dr. Lothar Thiele for supervising and guiding my research. His constant support was always very motivating, and the many fruitful discussions inspired much of the work presented in this thesis. I also appreciate very much that, even in the busiest moments, he had always time to talk about aspects of this thesis whenever I approached him.

I would also like to thank Prof. Dr. Koen Langendoen for being my co-examiner in this thesis.

I furthermore thank Dr. Jan Beutel for the many constructive discussions, his valuable suggestions, and for his help especially in the end of this work; Dr. Philipp Blum for the fruitful collaboration on the DSN Case Study and the interesting discussions about industrial applicability; Prof. Dr. Marco Platzner and Dr. Christian Plessl for the many inspiring discussions about reconfigurable computing; as well as Kevin Martin and Mustafa Yücel for the fruitful collaboration during and after their Master's Thesis.

I would also like to thank all my current and former colleagues of the whole TEC group for their company and support.

Finally, my dearest thanks go to Gabriela Hofer, her family, and my parents for their love and support throughout all these years of my education.

The work presented in this thesis was supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322. This support is gratefully acknowledged.

Contents

Abstract i

Zusammenfassung iii

Acknowledgement v

1 Introduction 1
  - 1.1 Validation Techniques for Embedded Systems 2
    - 1.1.1 Simulation 2
    - 1.1.2 Estimation 2
    - 1.1.3 Rapid Prototyping and Emulation 3
    - 1.1.4 System Test 3
    - 1.1.5 Formal Methods 4
  - 1.2 Visibility, Realism, Scalability and Effort 4
  - 1.3 Aim of the Thesis 5
  - 1.4 Contribution 6
  - 1.5 Thesis Overview 8

2 Performance Estimation for Design Space Exploration 11
  - 2.1 Wearable System Design 12
    - 2.1.1 Related Work 14
  - 2.2 Overview of the Exploration Methodology 14
    - 2.2.1 Problem Specification 15
    - 2.2.2 Architecture Model 16
    - 2.2.3 Exploration Environment 17
  - 2.3 Performance Estimation 18
    - 2.3.1 Usage Profile Specification 18
    - 2.3.2 Information Flow 22
    - 2.3.3 Physical Constraints 23
    - 2.3.4 Resource Specification: Computing Devices 24
    - 2.3.5 Resource Specification: Communication Channels . 27
  - 2.4 Exploration Environment 30
    - 2.4.1 Problem Specific Model 31

2.4.2 Architecture Representation 31
2.4.3 Architecture Evaluation 32
2.4.4 Architecture Selection 34
2.5 Exploration Results 34
2.5.1 Case Study Description 34
2.5.2 Results 36
2.5.3 External Module Extension 38
2.5.4 System Robustness 39
2.6 Summary 39

3 FPGA Emulation and Virtualization of Hardware Tasks 41
3.1 REnode – A Reconfigurable Embedded Node 42
3.1.1 Architecture Overview 43
3.1.2 Embedded Reconfigurable Hardware 43
3.1.3 Dynamic and Partial FPGA Reconfiguration 44
3.1.4 Hardware Tasks 45
3.2 Virtualized Execution of Process Networks 46
3.2.1 Virtualized Execution - Related Work 47
3.2.2 Task-Level Virtualization 48
3.2.3 Task Coordination Language 49
3.2.4 REnode Run-Time System 51
3.2.5 Restrictions and Extensions to the Coordination Language 57
3.2.6 Emulation based Validation 58
3.3 Partial Reconfiguration Design Flow 62
3.3.1 Direct Bitstream Manipulation 62
3.3.2 Bitstream Generation and Manipulation Tools . . . 63
3.3.3 Combining High-Quality Synthesis with Bitstream Manipulation 64
3.4 Case Studies and Prototypes 66
3.4.1 Reconfigurable Coprocessor 66
3.4.2 PDA–FPGA: Reconfigurable Embedded Machine . 67
3.5 Discussion and Summary 68

4 Distributed Algorithms 71
4.1 Practical Topology Control 73
4.1.1 The Topology Control Problem 73
4.1.2 Bluetooth Networking 74
4.1.3 BTnut Connection Manager 75
4.1.4 Topology Control — Related Work 76
4.2 Tree Algorithm 77
4.2.1 Tree Algorithm Validation 79
4.3 S-XTC: A Signal-Strength Based Topology Control Algorithm 81

4.3.1 Application Scenario 83
4.3.2 XTC Algorithm 84
4.3.3 Signal-Strength Measurements 85
4.3.4 S-XTC Algorithm 88
4.3.5 Wireless Communication 89
4.3.6 Dynamic Adaptation 90
4.3.7 Bounded Degree 93
4.3.8 Improved Scalability in Dense Networks 97
4.4 Case Study 99
4.5 Summary and Discussion 100

5 Deployment Support Network 103
5.1 Development Support and Testing of WSNs – Related Work 104
5.2 Deployment Support Networks 106
5.2.1 DSN–Architecture 106
5.2.2 DSN–Services 108
5.2.3 Test Automation 112
5.3 Realization 112
5.3.1 Bluetooth Scatternets 113
5.3.2 Wired Target Interface 114
5.3.3 Client Interface 115
5.3.4 Performance Evaluation 115
5.4 Case-Study: Link Characterization in Buildings 117
5.4.1 Experiments 117
5.5 Summary 119

6 Conclusions 123
6.1 Summary of Contributions 123
6.2 Future Directions 124

Bibliography 126

Acronyms 139

Curriculum Vitae 140

1

Introduction

Todays, the world is full of embedded systems. In an average car for example dozens of small microprocessors control the stearing, the breakes, the airbags and many other parts. These microprocessors are embedded, i.e. they have sensors as input (e.g. the stearing wheel) and they control actuators (e.g. the breakes). There are many different kinds of embedded systems. Some are simple like a washing machine, others more complex such as airplanes or an industrial plant. We speak of a *distributed* embedded systems, when it consists of individual components that are distributed in space and time. In other words, they are not connected to a central controller and they typically do not share a common time base.

A wireless sensor network (WSN) is an example of a distributed embedded system. It is a computer network consisting of spatially distributed autonomous devices using sensors to cooperatively monitor physical or environmental conditions, such as temperature, sound, vibration, pressure, motion or pollutants, at different locations [RM04]. The devices in a WSN, the sensor nodes, are typically small and battery operated, and consist of a radio, sensors, and a minimal amount of on-board computing power.

Sensor nodes are deeply embedded into the environment. They are physically distributed and communicate over an unreliable wireless channel. Algorithms must be executed efficiently in terms of energy on tightly limited resources. This all makes the hard- and software design of WSNs a very difficult task.

An essential aspect of embedded system design is validation. Validation is the process of checking whether or not a certain (possibly partial) design is appropriate for its purpose, meets all constraints and will perform as expected [Mar03]. Validation is important especially for WSNs, because the involved uncertainties make it hard in practice to design correct implementation from the specification.

To verify only at the very end of the design process normally does not work, since there are large differences between the level of abstraction used for the specification and that used for the implementation. As a consequence, validation is required at different steps in the design flow.

For the different phases in the design process and the different levels of abstractions different validation strategies are required. There is no single strategy available that solves all problems.

1.1 Validation Techniques for Embedded Systems

In this section we will provide a brief overview over the key techniques for embedded system validation.

1.1.1 Simulation

In simulations, design models are executed typically on general purpose computers. The design model abstracts from the real design. Choosing the right abstraction is always a compromise between simulation speed and accuracy. Faster simulations provide less accuracy.

During the last decade, a number of simulators with different level of abstraction have been developed that are tailored to the domain of e.g. wireless sensor networks. Prominent examples are NS-2 [ns2], Glomosim [ZBG98], and TOSSIM [LLWC03].

Simulations are a very common technique for validating designs and algorithms, since the execution can be easily controlled and observed by a developer. Furthermore it is possible to simulate a large number of nodes without the sensor node hardware.

1.1.2 Estimation

Estimation is the calculated approximation of a result which is usable even if input data may be incomplete, uncertain, or noisy. The models used in estimation are usually simpler than the ones used in simulation. The power consumption of a sensor node can for example be estimated considering only the duty-cycle of the nodes. The result of this approximation is of course less accurate as a simulation with a fine-grained power-model such as PowerTOSSIM [SHC$^+$04]. However, the advantage of estimation is that the calculation is much faster and it does not require detailed input data. This is important for design space exploration in an early phase, where a number of design alternatives have to be quickly evaluated.

1.1.3 Rapid Prototyping and Emulation

Emulation and simulation are similar in the sense that the design is not evaluated on real hardware. The difference is that emulators are more accurate because they execute actual application code on a platform which behaves "almost" like the final system.

Emulation is often applied in prototyping. E.g. a prototype for an ASIC device can be built using FPGAs. These prototypes should essentially behave like the final systems, but they may be larger, consume more power, and have other properties which are acceptable for evaluation. Emulators are more transparent than the final system, since developers can better access internal state information. In the case of FPGA-based emulators, they are also more flexible in the sense that the circuit is reprogrammable which allows the developer to rapidly evaluate different solutions. This type of emulation is called hardware emulation.

The definition of software emulation is not as clear as of hardware emulation. There exist a large number of systems that are referred to as emulators such as CPU-, terminal-, printer-, or game console emulators. They have in common that they have the ability to *imitate* another program or device. However there is no clear distinction between simulation and emulation. The two terms are mostly interchangeable in many cases.

The imitation of a processor is referred to as CPU emulation or also as CPU simulation. There exist a number of CPU emulators for microcontrollers that are typically found on sensor nodes. Such emulators have been enhanced with the functionality of emulating simultaneously multiple networked devices. Examples are ATEMU [PBM$^+$04] and Avrora [TLP05].

1.1.4 System Test

During the design of embedded systems, testing is usually done for every layer of abstraction. By applying so-called *test-patterns* as input, we can observe the behavior of individual components, or the complete system and compare it with the expected response.

Testing is essential for wireless sensor networks. Simulation and software emulation are not sufficient as they cannot capture the complex physical phenomena that appear in real deployments. The influence of effects such as interference and multi-path fading must be tested with a deployment in a realistic environment.

The deployment of a sensor network can be a major effort, especially in the case of an outdoor application that comprises a large number of nodes. Furthermore, testing is usually done not only once, but several times and the nodes eventually must be updated with new software. Without special infrastructure, testing sensor networks in a real deployment is very time consuming. Therefore, researchers have built dedicated sensor network testbeds that consist of a fixed number of nodes. The nodes are placed at fix locations in a lab, typically arranged in a grid. The nodes are additionally wired to a central PC in order to reprogram the nodes and to collect the test output data. Examples of such testbeds are MoteLab [WASW05] and the Ceiling-Array of Emstar [EGE04]. These testbeds are often also referred to as *emulation*-testbeds, as the application is tested on a platform that behaves "almost" like the actual system.

1.1.5 Formal Methods

Validation with formal methods is concerned with formally proving a system correct, using the language of mathematics. In order to make formal methods applicable, a *formal model* is required. For WSNs, not only models of computation (e.g. process networks, petri-nets, state-charts) are used, but also visibility models (e.g. unit disk graph), deployment models (e.g. uniform or random), and mobility models (e.g. random waypoint).

Formal models are also applied in simulations. However, one difference between simulations and the validation using formal methods is that the results obtained through simulation depend on the input patterns of a simulation run. Furthermore, simulation can only prove the existence of a certain property in simulation runs. With formal methods, one attempts to prove these properties mathematically.

1.2 Visibility, Realism, Scalability and Effort

The validation strategies presented above can be applied to the domain of distributed embedded systems. We can compare the different strategies and discuss the advantages and limitations using four criteria:

Visibility: With *visibility* we denote the amount of information of the validated system that the developer can access. Visibility is important, when one is not only interested in detecting, but also in localizing errors. Applied to WSNs, visibility is for example the access to the state of the distributed nodes or to the state of the network.

Realism: Not only the amount of state information is important, but also the quality. Methods which use very high abstractions provide less realism than methods which can be applied to real deployed systems.

Scalability: This measure indicates how well a strategy scales to a large number of nodes.

Effort: With *effort* we denote the time and the additional infrastructure that is needed to validate a system.

Simulation and estimation generally provide excellent visibility, since validation is done in a controlled environment such as a single PC. However there is a clear trade-off between realism, scale and effort. As an example, there exist a large number of simulation models for wireless communication with different levels of abstraction. Some fine-grained models can simulate the communication with high realism incorporating interference and fading effects. However, only for a very limited number of nodes. For scenarios with more nodes, less complex models are required in order to keep the resource usage and simulation time practical.

Indoor emulation-testbeds for WSNs provide higher realism than simulators because the application is executed on the real sensor node platform. But there is still a large gap in realism when compared to a real world deployment. The reason for this is that in testbeds the system is validated in an artificial environment. Sensor nodes are per definition tightly integrated into their environment. Small changes in the environment can change the behavior of the system significantly. One example is the wireless communication that is very sensitive to the presence of interference.

A field-test would provide the required realism. However, field-tests are very difficult to conduct. In order to achieve the minimal visibility that is required for validation, an enormous effort is needed.

1.3 Aim of the Thesis

Validation is an integral and indispensable part of the design flow for both hard- and software based systems. It is the only way to relate requirements and specification to the real system. Validation is further a mean to obtain objective quality metrics about systems in their target environment and, not less important, to gain confidence in the systems quality. Validation is also important from an economic viewpoint, as the error correction costs dramatically increase the later the errors are detected.

Validation is difficult. This can be seen when looking at the time spent on validation: For software based systems, it is estimated that about 50 percent of the complete design time is spent on testing. The validation of distributed embedded systems is even more complex. Some of the reasons are (a) unreliable communication infrastructure, (b) distributed nature of the system, (c) limited resources, and (d) limited infrastructure for inspecting the system.

The validation of distributed embedded systems is an interesting problem which requires new research, i.e. new strategies are needed that improve visibility, realism, scalability and effort.

In this thesis we address this challenge by investigating different levels of abstraction and applying validation strategies that are adapted to the peculiarities of distributed embedded systems.

1.4 Contribution

In this thesis we make a number of contributions to the state of the art in validation and design of distributed embedded systems. In order to achieve our goal, we present a vertical slice of the design space concerned and discuss relevant questions encountered in different phases and levels of abstraction in the design and development process.

The main contribution are the following:

1. We present a new methodology for evaluating promising candidate architectures in an early design phase in the case of multi-module wearable systems. The evaluation is based on the estimation of computing performance, communication delay and the overall system power consumption. As a further objective our method estimates the flexibility of the system which depends on the weight and size of its components and the type of communication that is used between individual modules. We have embedded the architecture evaluation into an automatic design space exploration environment that uses multi-objective optimization and evolutionary algorithms.

2. We have built an FPGA-based emulation framework that emulates a coprocessor that is reconfigurable at runtime. The coprocessor can be considered as a hardware task. We present a run-time system for resource-constrained embedded nodes that is able to execute applications consisting both of hardware and software tasks. The coordination of the tasks is described in a formal language which is interpreted on a CPU. This abstraction allows for the validation of individual tasks and the performance evaluation of the run-time

system using simulation and emulation for different configurations. We have further built two prototype implementations that proof the functionality of the concept.

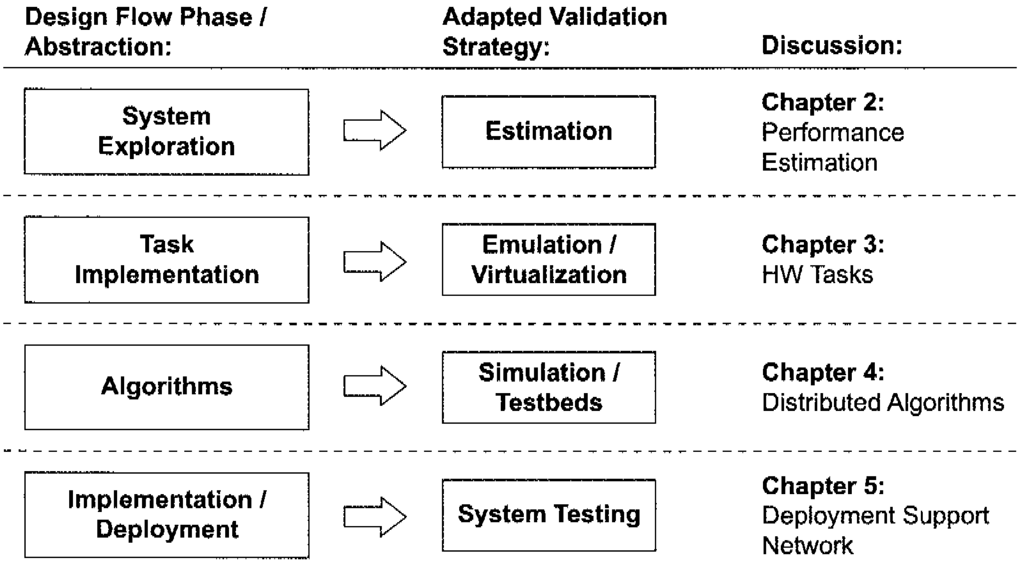
3. Many proposed wireless sensor network algorithms have never been implemented and validated on real sensor node platforms. In most cases, only simulation results are available that are often based on impractical or unrealistic assumptions. We present a complete implementation of a distributed topology control algorithm for wireless sensor networks. We extended an existing algorithm with additional heuristics in order to make it practical. We validated our algorithm with formal analysis, simulation and measurements. We have further developed a modular framework that allows for testing topology control algorithms on a large number of real sensor nodes. We have implemented two different topology control algorithms on the BTnode platform and tested them with over 50 nodes. By comparing measured with simulated results we provide valuable feed-back to algorithm designers.

4. We present the deployment support network (DSN), a new methodology for testing wireless sensor networks in a realistic environment. We suggest to deploy the WSN under development along with a second wireless network, the DSN, with DSN nodes that are attached to the original sensor nodes. The DSN nodes use topology control in order to form a reliable wireless backbone network, which is used to transport the test- and control data to and from a central server. We describe the different available services for testing WSN applications. We compare this methodology with other approaches such as testing on WSN testbeds and suggest that our method is more flexible, platform independent and returns high quality information because it allows for testing the applications in a realistic environment. We present a case study in which our methodology was applied by an industrial partner for the automated validation of their wireless product.

The major contributions presented in this thesis have been published in the following refereed publications [DPP02, ABD+04, DPT04, BDH+04, BDMT05, DBM05, NBD06, DBT07a, DBT+07b].

1.5 Thesis Overview

In the following we give an overview over the contents of this thesis (see also Figure 1.



Design Flow Phase / Abstraction:	Adapted Validation Strategy:	Discussion:
System Exploration	Estimation	**Chapter 2:** Performance Estimation
Task Implementation	Emulation / Virtualization	**Chapter 3:** HW Tasks
Algorithms	Simulation / Testbeds	**Chapter 4:** Distributed Algorithms
Implementation / Deployment	System Testing	**Chapter 5:** Deployment Support Network

Fig. 1: Thesis overview.

- In **Chapter 2** we describe the performance estimation of a distributed embedded system. We present a design space exploration method for exploring different architecture configurations. We discuss the overall exploration framework and describe the specification of the application scenario, the representation of a single design point and its evaluation where the fitness of the candidate architecture is estimated for multiple objectives.

- **Chapter 3** describes reconfigurable hardware components and their virtualized execution on an FPGA based emulation environment. Applications are described as a collection of hardware tasks and a formal description of the task coordination which is based on the process network model. This virtualized execution allows for the deployment of new applications on distributed nodes.

- In **Chapter 4** we discuss topology control algorithms. We analyze typical assumptions that are made in algorithm descriptions and simulation models and validate the practicability on real resource-constrained sensor nodes.

- **Chapter 5** introduces the *Deployment Support Network* as a new validation strategy for sensor networks. We describe the concept and its functionality and discuss the gain of quality and realism in the design process. We further present a case-study that has been conducted at an industrial partner, who used our prototype implementation of the DSN for testing his products.

- **Chapter 6** concludes the thesis with an outlook for future research and a summary of the contributions.

2

Performance Estimation for Design Space Exploration

In this chapter, we present efficient performance estimation applied to the design space exploration of wearable systems. A wearable system is a distributed embedded system where energy consumption, size and weight are important factors that need to be minimized.

As described by Weiser [Wei91] and Mann [Man98] a wearable computing system can be seen as an active extension of the user, enhancing his intelligence, augmenting his ability to communicate and interact with the environment, and assisting him in a variety of everyday situations.

Wearable systems need to have many properties that make them significantly different from a conventional mobile machine (e.g. a laptop or a PDA). In terms of functionality this includes situation and context awareness, the ability to act proactively rather than just react to explicit user commands, the ability to overlay complex information over the user's view of the reality, a high degree of connectivity and a sophisticated user interface that allows the system to be used while mobile. From the hardware point of view, three issues are of particular importance. First, the system has to execute widely varying computational loads with adequate speed while coping with much stricter power consumption constraints than most standard mobile systems. Second, it has to combine sensors and input/output (IO) devices placed at different locations on the user's body into a distributed heterogeneous system, e.g. a display in the glasses and a motion sensor on the wrist. Finally, a wearable system needs to be unobtrusive to the degree that it does not interfere with the user's activity and does not change his appearance in any unacceptable way.

Implementing and combining those properties into a working system poses many challenges. In this chapter, we focus on the architecture evaluation by adapting performance estimation to distributed wearable architectures. Estimating system performance and properties of such systems is difficult, because of the varying workloads that are executed on distributed, heterogeneous computation and communication modules.

In the proposed methodology, we address this challenge by modeling the complete wearable system including the workload, the application scenario, and the computation and communication resources. We then evaluate architectures and scenarios using the models and a analytic estimation of the system performance.

We apply estimation and not simulation for the validation. This has the advantage that we can evaluate various solutions in short time, which is important if we want to explore the large design space.

In Section 2.1 we introduce design goals and constraints that appear in wearable systems and discuss related work. Section 2.2 outlines the exploration environment. In Section 2.3 we discuss the performance estimation for different objectives. Section 2.4 explains the multi-objective optimization process that produces a set of Pareto-optimal solutions. In Section 2.5 we discuss the results of a case-study and summarize the chapter in Section 2.6. This chapter is based on [ABD+04].

2.1 Wearable System Design

The design space of a wearable system can be described by the following, mostly conflicting global goals:

- *functionality*, providing as much of the wearable features as possible together with task-specific functionality in an efficient and user-friendly manner;
- *battery lifetime*, making the system constantly operational without the need to change or recharge batteries for as long as possible by minimizing power consumption; and
- *wearability*, comprising a variety of ergonomic criteria including size, weight, correspondence between shape and placement on the body, radiation concerns, heat and aesthetic issues that are necessary for an unobtrusive system implementation.

While aiming to achieve these goals the design needs to take four types of constraints into account:

1. *Usage profiles* that specify the required functionality and the relative importance of different features. The vision of an intelligent personal assistant implies a variable, dynamic usage profile with changing context-dependent applications.
2. *Information flow* given by the necessity of placing IO devices and sensors at different locations of the body, which implies a distributed, heterogeneous system architecture.
3. *Physical constraints* that provide ergonomic constraints on the weight and placement of the components on different body locations as well as the relative importance of the wearability criteria at different locations.
4. *Hardware resources* available for implementation that are determined by the state-of-the-art technology as well as cost, compatibility and other strategic concerns.

In summary, the design of a wearable architecture can be viewed as a *multi-objective optimization problem.* For a known—but highly dynamic—context-dependent usage profile, it aims to find the optimal assignment of computation and communication resources to a number of computing modules distributed over the user's body.

For each module of such an architecture, a choice must be made between providing it with enough intelligence to perform computations locally or sending away raw data for processing on other resources. Furthermore, each module must combine energy-efficient execution of some permanently running low-intensity sensor monitoring and evaluation tasks with the high computing power required by occasional performance bursts. As a consequence, it can be necessary—even inside individual modules—to implement heterogeneous, dynamically configurable systems, as for example proposed by Plessl et al. [PEW+03].

The dynamic nature of the usage profile implies modeling by means of an abstract workload characterization rather than by means of particular applications. Such a workload characterization must focus on the temporal variations of the required computing performance and the communication pattern.

By incorporating these aspects, the design of a wearable system architecture involves the selection of modules, their components and appropriate communication channels. That way, the system performance as prescribed by the usage profile is optimized with respect to power, execution speed, a set of specialized wearability criteria, and cost. The problem is not unlike design automation tasks from the embedded systems domain, for which various automatic design methodologies have been developed. However, to the best of our knowledge, no attempt has been made to apply such methodologies to the specific problem of wearable computing architectures.

2.1.1 Related Work

Most wearable systems are based on conventional notebook architectures integrated into some sort of belt or backpack harness. For some purposes, in particular in well defined industrial applications, such designs are justified and have proved to be successful tools [SS99]. Smailagic et al. [SRS00] have proposed a systematic design process for such systems.

When it comes to realizing the vision of a wearable computer as a context-aware, proactive and intelligent personal assistant, such traditional architectures are only of limited value. As Baber et al. [BHW99] have suggested, distributed and heterogeneous systems consisting of a mixture of low-power general-purpose processors, signal processors and special-purpose circuits seem a more promising approach. While this view is shared by many in the community, few attempts have been made so far to model, evaluate and implement such systems. In particular, except for the evaluation of the power consumption of individual devices [SK97], there are no quantitative results documenting under what circumstances the distributed, heterogeneous approach actually outperforms classical centralized architectures.

There are various methods available to explore the design space of computer architectures. However, only a few references will be given here as it is not the purpose of the work to advance the state of the art in this area in general. Many known approaches to the design of architectures deal with heterogeneous systems consisting of different sorts of components, e. g. [Wol02, BTT98, Gup95, KC98, LCBK01], or with communication aspects, e. g. [LRD01]. Some of them particularly deal with conflicting criteria in the optimization process [Mic94, ETZ00]. However, such methods have not been applied to the design and evaluation of distributed wearable systems so far. Thus, it is not clear how the peculiarities of distributed wearable systems can be taken into account. What especially needs to be investigated is, how existing methods can be extended and applied in order to deal with dynamically varying, context dependent usage scenarios.

2.2 Overview of the Exploration Methodology

The goal of the exploration methodology is to assist the designer of a wearable system in determining which configurations of heterogeneous com-

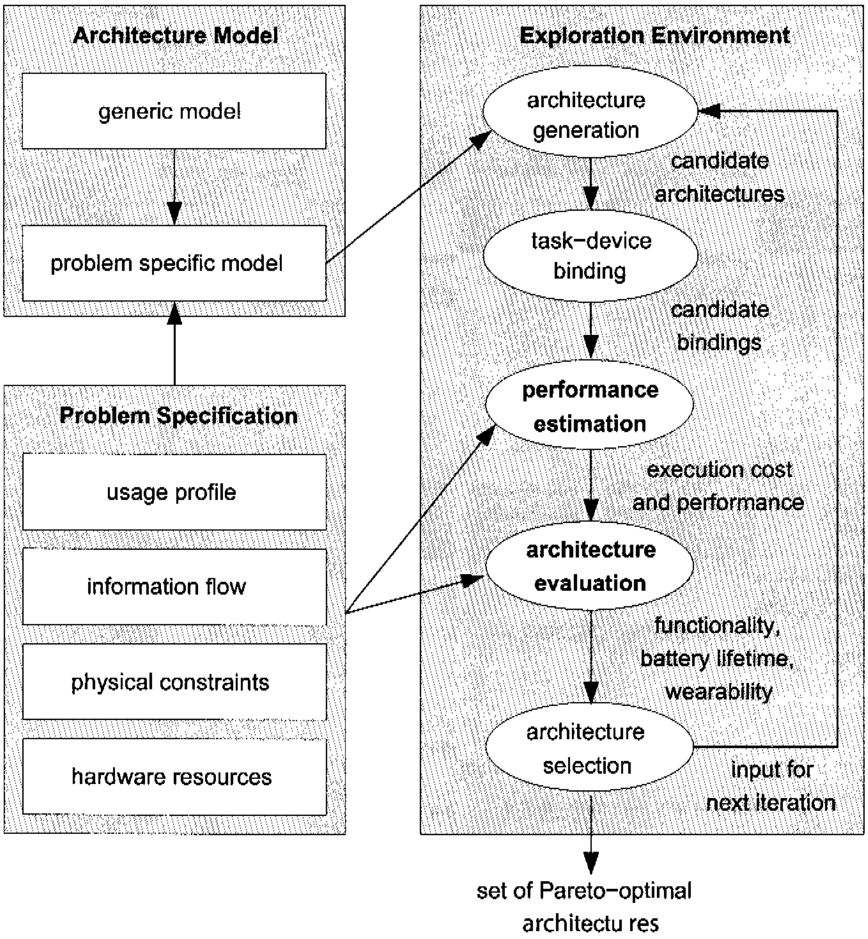


Fig. 2: Modular exploration methodology consisting of three main components: problem specification, architecture model and exploration environment.

puting and communication resources distributed over the user's body are most suitable for a given problem. The major parts of the exploration methodology are the analytic models, the performance estimation, and the architecture evaluation. Fig. 2 shows an overview with three components: the *problem specification* of a particular design under investigation, the *architecture model*, which spans the space of possible solutions and the *exploration environment*, which conducts the search and derives the architectures best suited to fulfill the specifications.

2.2.1 Problem Specification

The problem specification defines the analytic models that are applied in the performance estimation and architecture evaluation. The description of a specific design problem involves four steps, which correspond to the four types of design constraints outlined in the introduction: usage profile specification, information flow specification, physical constraints specification, and hardware resource specification.

The *usage profile specification* characterizes the desired functionality through a statistical description of the expected variations of the computational load and the communication pattern over time. In essence, it contains a hierarchical set of task graphs together with hard and soft timing constraints. This provides a temporal distribution of the amount and type of computation that the system needs to perform. The specification also includes data flow patterns, which determine the inter-module communication load arising when different parts of the computation and the input/output operations are performed on different modules.

The *information flow specification* assigns a set of body locations to each input and output related task of the usage profile specification where the task can be executed.

The *physical constraints specification* defines the wearability criteria and their relative importance. The wearability of a system depends on many factors ranging from such obvious and easily quantifiable aspects as size and weight to more subtle issues such as health concerns (e. g. related to EM radiation or heat dissipation) or aesthetic considerations. In general, the choice of the relevant factors also depends on the particular application and on the locations at which the modules are placed. To be able to flexibly accommodate a wide range of different criteria, we use a problem specific *wearability factor.*

The *hardware resource specification* provides a set of computation and communication channel devices available for the design. The specification includes formulas to calculate the power consumption for different types of computation and communication load, as well as values for the measures that are used to calculate the wearability factor.

2.2.2 Architecture Model

The architecture model comprises the main interface between the problem specification and the exploration environment. It consists of the *generic model,* which describes the overall types of architectures considered by our methodology, and the *problem specific model,* which incorporates the design constraints of the problem in question.

Fig. 3 shows the generic model consisting of a set of computing modules distributed over the user's body. Each module contains devices and communication channel interfaces. The devices can be processors, application-specific integrated circuits (ASICs), sensors or IO interfaces. For *inter-module communication* there exists a set of connections. Each connection consists of one or more physical channels matching the channel

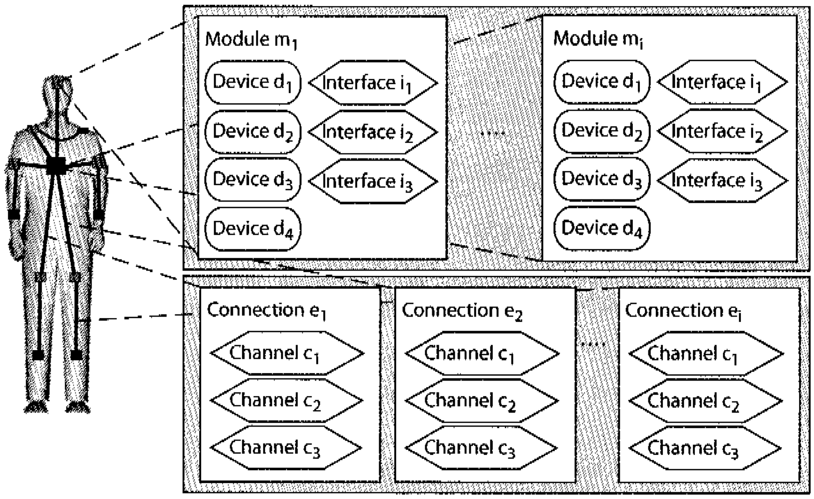


Fig. 3: The generic system model consists of distributed, partly connected modules containing all possible resources, devices and communication channel interfaces. The connections comprise a set of available physical channels.

interfaces of the corresponding modules.

The problem specific model is derived by combining the generic model with the problem specification. It first defines the system topology by specifying a particular subset of modules and connections that are to be used. It also specifies the devices that the modules can contain, as well as the channels that can be used for the connections.

2.2.3 Exploration Environment

As Fig. 2 shows, the performance estimation is embedded into an iterative search process that is conducted in five steps. First, the design points to be visited in the search space are determined by deriving a number of candidate architectures from the problem specific model. The tasks specified by the usage profile are then bound to specific devices. In the third step, the performance (in terms of execution time and communication delay) and the execution cost (in terms of power consumption) are estimated. In the next step, the three optimization criteria, i. e. functionality, battery lifetime and wearability, are evaluated. Finally, a decision is made on how to proceed with the search. The search is either terminated or the results are passed on to the first step as a starting point for the selection of the design points to be visited next in the search space.

The modular concept of the exploration environment allows to use different search and evaluation algorithms as well as wearability measures. In particular, it is possible to employ other scheduling and load estimation algorithms developed in the parallel and distributed computing community in the 'task-device binding' and 'performance estimation' modules, e. g. based on statistical analytical models, simulation or trace-based approaches, e. g. [LRD01].

To employ an automatic design space exploration, the optimization criteria must be formulated in quantitative terms:

- The *functionality* is defined by the usage profile specification. We assume that all valid architectures are able to provide the functionality. However, the architecture has an impact on the execution and communication delay. We thus use the delay as functionality criterion.
- For the *battery lifetime* to be independent of the particular battery type, we use the average system power consumption as a quantitative measure.
- As a measure of a system's *wearability*, a weighted sum of the wearability factors of all components is used. This allows a flexible inclusion of different criteria while providing a single quantitative value, which can be easily handled in the optimization process.

2.3 Performance Estimation

2.3.1 Usage Profile Specification

The usage profile specification intends to capture the workload characteristics of the wearable system. This includes the variation of the computation intensity as well as the spatial distribution of computation and communication. The specification model is hierarchically structured into tasks t, applications a and scenarios s. Fig. 4 illustrates an example comprising three scenarios. The tasks constitute the atomic units of computation and communication. A set of tasks is assembled into an application. A scenario contains a set of applications that run concurrently on the wearable system within predefined hard and soft timing constraints. We assume that at any given time, exactly one scenario is active. This implies that a change in the state of the wearable, e. g. when responding to a user request, causes a new scenario to become active. With regard to a given usage profile, the percentage devoted to a particular scenario is specified by the scenario weight W_{scen}.

2.3.1.1 Tasks

A task t is defined as a self-contained unit of computation that is characterized by three parameters: the amount of input data, the computational

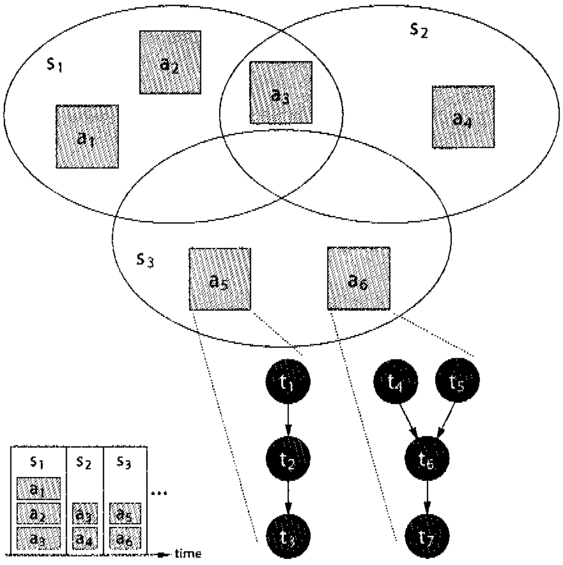


Fig. 4: Example of a usage profile with three scenarios s_1, s_2 and s_3. Each scenario comprises a set of concurrently running applications: $s_1 = \{a_1, a_2, a_3\}$, $s_2 = \{a_3, a_4\}$, and $s_3 = \{a_5, a_6\}$. Each application contains a set of tasks, e. g. $a_5 = \{t_1, t_2, t_3\}$ and $a_6 = \{t_4, t_5, t_6, t_7\}$, represented by a DAG. At any given time, exactly one scenario is active.

load and the amount of output data. There are no restrictions on the size or complexity of a task. Thus, a task can be a small signal processing kernel, a simple utility, or a complex computation.

To characterize the computational load of a task, we use its instruction mix. The technique and the characterization of applications presented here are the results of the work conducted by Enzler [Enz04].

The instruction mix quantifies the amount and types of instructions required by a task to process the input data. The instruction mix depends on the algorithm, the input data, the compiler, and the processor's instruction set, but is independent of any architectural parameters of the processor such as the number of execution units, cache sizes, or the like. To simplify the estimation, we have grouped the instructions into seven representative classes:

- *int-cheap* integer instructions (logic, shift, addition, subtraction, comparison),
- *int-costly* integer instructions (multiplication, division),
- *fp-cheap* floating-point instructions (addition, subtraction, comparison, miscellaneous),
- *fp-costly* floating-point instructions (multiplication, division, square root),
- *load and store* instructions,

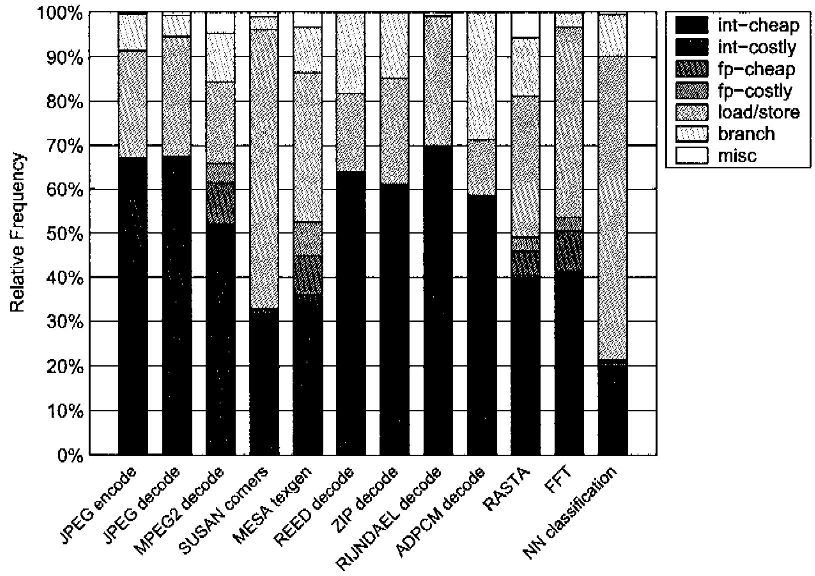


Fig. 5: Instruction class mix of some selected tasks, depicting the relative instruction class frequencies during execution.

- *branch* instructions, and
- *miscellaneous* instructions.

Every task t is assigned an *instruction class mix vector* $\vec{I}_t$, which contains the number of instructions of each instruction class.

We gather the instruction class mix by means of the processor simulation tool set SimpleScalar [ALE02]. SimpleScalar's processor model is based on a RISC architecture with a MIPS-like instruction set. Using the *sim-profile* statistical profiling mode, we determine the total number of executed instructions as well as a detailed breakdown of the instruction frequencies during execution. We have chosen a set of tasks from typical application domains in order to represent a characteristic workload of a wearable system. We have simulated all tasks using out-of-the-box code, i. e. code without manual optimization. Since the original data sets of the programs are often small and serve only for test purposes, we have chosen our own input data sets. Fig. 5 depicts the instruction class mix for some of the investigated tasks. Table 1 lists the used input and output data as well as the measured instruction counts.

2.3.1.2 Applications

We define an application a to be a set of tasks t in the form of a directed acyclic graph (DAG). We denominate the source nodes of the DAG as input tasks and the sink nodes as output tasks. An application consists of at least three tasks: one input task, one or more computational tasks

Application	Input Data	Output Data	Instruction Count	Timing Constraints (R or D_{max})	Comp. Power Requirements [MIPS]
JPEG encode	1280x1024 image	1280x1024 image	238,308,913	0.1 – 10 s	24 – 2,383
JPEG decode	1280x1024 image	1280x1024 image	159,315,874	0.1 – 1 s	159 – 1,593
MPEG2 decode	4-frame movie	4 raw frames	182,888,390	4 – 7.5 Hz	732 – 1.372
SUSAN corners	784x576 image	corner coordinates	222,384,178	0.05 – 1 s	222 – 4,447
MESA texgen	500x500 image	500x500 image	75,689,503	0.03 – 0.1 s	757 – 2,271
REED decode	414k binary	362k text	451,744,148	0.1 – 1 s	452 – 4,517
ZIP decode	491k binary	2610k text	68,643,500	0.1 – 1 s	68 – 686
RIJNDAEL decode	362k binary	362k text	32,928,120	0.1 – 1 s	33 – 329
ADPCM decode	1 sec. audio	1 sec. audio	311,098	1 Hz	0.3
RASTA	1 sec. audio	4.7k binary	18,254,847	1 Hz	18.3
FFT	5k samples	5k samples	5,096,956	0.5 – 5 Hz	2.5 – 25
NN classification	1 pattern (8 values)	1 decision value	9,785	1 – 10 Hz	0.01 – 0.1

Tab. 1: Characterization of some selected applications. Columns 2–4 specify input and output data and the measured number of executed instructions. Column 5 shows the assumed timing constraints (depending on the application either the repetition frequency R or the maximum latency D_{max}). Column 6 lists the derived computing power requirements

and one output task. This representation allows to model the computation/communication trade-offs involved in distributing the execution of an application onto different devices. For this purpose, the input and output tasks are treated in a special way: their computational loads are defined to be zero. In this manner, input and output tasks may be assigned to the wearable's IO devices while the computational tasks are assigned to the computing devices. This reflects the fact that many IO devices, e. g. sensors, do not have any computation capabilities. However, if an IO device, e. g. a smart sensor, provides computation capabilities, computational tasks can be assigned to it as well.

2.3.1.3 Scenarios

The task specification determines the amount of computation and communication that needs to be performed on the wearable system. In a particular scenario s, these figures are translated into computation and communication requirements by assigning two timing parameters to each application: The repetition frequency R and the maximal latency D_{max} acceptable for execution.

Typically, wearable systems do not feature such stringent real-time constraints as e. g. embedded control systems. The real-time constraints of a wearable system rather come from applications that require a continuous, periodic processing pattern. Examples for this are the evaluation of context sensor data or the processing of audio and video frames. Our specification model incorporates such real-time constraints via the repetition frequency R. However, many applications are not real-time but latency constrained. A user request satisfied within about 100 ms is usually perceived as instantaneous [SN97]. The latency constraints are thus an issue of user preferences. The system's user friendliness can be treated as a soft optimization criterion.

Fig. 6 outlines the computing power requirements of some of the investigated applications. These performance figures have been gathered by combining the measured numbers of executed instructions of the tasks with assumed values for R or D_{max} as listed in Table 1. The results show the diverse requirements of the different application types. Note that the ranges have been intentionally set large to cover a wide variety of scenarios.

2.3.2 Information Flow

The information flow specification is based on the fact that every possible input/output signal is associated with a particular input/output task of the usage profile, e. g. whenever output is required on a display, the

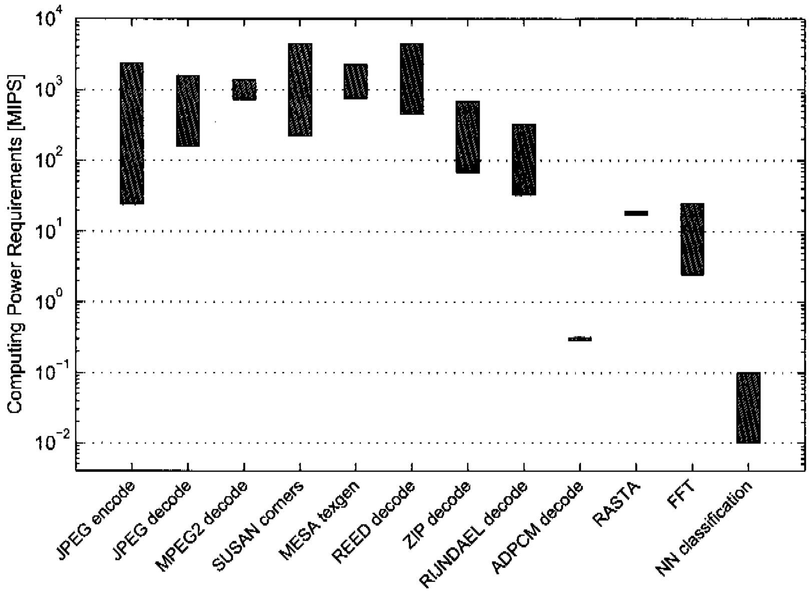


Fig. 6: Computing power requirements of some selected applications.

corresponding application contains an appropriate output task.

The information flow specification starts with a set of possible locations $\mathcal{P} = \{p_1, p_2, \ldots\}$, which are relevant for a particular application. Each task t is then assigned a subset of locations $\mathcal{P}^t \subseteq \mathcal{P}$ where the task could potentially be executed.

2.3.3 Physical Constraints

Physical constraints reflect wearability considerations that determine how obtrusive the system appears to the user. Obvious factors to be considered are the size, weight, EM radiation emission and heat dissipation of the system. However, depending on the application, the user's personal preferences and the location of the system components on the body, different factors might be relevant and/or they might be given a different level of importance.

Deciding which factors are important for the wearability of a system is up to the ergonomic research and social acceptability studies and is not discussed any further in this work. Instead, we focus on providing a flexible mechanism for the inclusion of different constraints and their respective importance in the architecture evaluation. This is done through a *wearability factor* and a *power weight vector*.

Wearability factor

The wearability factor is calculated by a *wearability vector* $\vec{w}$ and a *wearability weight matrix* W. There is one wearability vector for each resource. Each element of the vector $\vec{w}$ is devoted to a different factor influencing the wearability. The elements of the matrix W specify the relative importance of the factors, addressed by the wearability vector for different body locations. It has one column for each location. The pth column $\vec{W}^p$ contains all weights for location p. The abstract wearability factor of a resource placed at a particular location is then given by the dot product $\vec{W}^p \cdot \vec{w}$. These abstract values are then used to compute the system wearability.

Power weight vector

Batteries and power generation devices or power transmission wires can be more or less burdensome depending on the location of the module. In addition, some body locations provide much better conditions for power generation than others. Large area solar cells for example can easily be placed on the outer, upper back surface of clothing. By contrast, energy generation in the glasses is rather difficult. To reflect these considerations, the *power weight vector* $\vec{w}_P$ specifies a set of location dependent, problem-specific weights for the power consumption of modules. If a module is placed at a location p then its power consumption is scaled by multiplying it with the pth element of the vector $\vec{w}_P$.

2.3.4 Resource Specification: Computing Devices

The resource specification describes the computing devices and communication channels, which are potentially incorporated into the system under investigation. The specification consists of three parts: (a) a set of parameters used to describe each device or channel, (b) formulas that in consideration of a particular workload translate these parameters into power consumption and delay values, and (c) a set of accordingly parameterized devices and channels. This section concentrates on the computing devices; the next section discusses the communication channels.

2.3.4.1 Specification Parameters

Each computing device d is characterized by the execution speed, some electrical parameters, and the wearability factor. Currently, the memory architecture and the memory energy consumption are not modeled. As an approximation we assume that the memory configuration is independent from the computing device configuration.

Execution speed

The execution speed specification consists of the clock frequency of a device and the number of execution cycles for the instruction classes. To accommodate the fact that many devices can operate in a frequency range, a minimum clock frequency f_{min} and a maximum frequency f_{max} are specified.

For each device d, the number of execution cycles of each instruction class is given by the *instruction class execution time vector* $\vec{I_D}$. The instructions classes correspond to the classification defined in the usage profile. The average execution time for each class is calculated from data sheet values. Non-implemented instructions are emulated by existing instructions. For superscalar devices, we follow the approach by Cvetanovic and Bhandarkar [CB96] in estimating an average number of cycles per instruction (CPI), which describes the speedup resulting from instruction parallelism. We assume a CPI of 0.83 for a 2-issue processor and a CPI of 0.58 for a 4-issue processor.

Often, field-programmable gate arrays (FPGAs) or ASICs are employed in applications with critical constraints on performance or power consumption. Since these devices may not rely on an instruction set, the instruction-based specification approach is not applicable. Instead, these cases can be considered at task level: a task implemented on an FPGA or ASIC can be incorporated directly as a task–device pair with simulated or measured execution time, energy consumption and wearability factor.

Electrical parameters

The electrical parameters characterize the power consumption of a device. We consider three operation modes: sleep, idle and execution. The specification consists of the sleep power P_s (sleep mode), the idle power P_i (idle mode), and the energy consumption per cycle E (execution mode).

The idle and sleep power figures are currently derived from data sheet values. For the execution mode, we rely on an approach proposed by Sinha and Chandrakasan [SC01], which employs instruction energy profiling. Based on their observations, we assume that the energy consumption depends only on the execution frequency of the instructions, but not on their type. The results reported in [SC01] show that for Hitachi SH-4 and SA-1100 devices the accuracy lies within 8%.

In order to take dynamic voltage scaling (DVS) into account, minimal and maximal energy consumption per cycle E_{min} and E_{max} are specified. E_{min} represents the energy consumption per cycle when running at the minimum cycle rate f_{min} and at the lowest possible voltage. Accordingly, E_{max} refers to the maximum frequency f_{max} and the required higher voltage. The energy consumption per cycle at frequency f is approximated with

$$E = E_{dyn} + E_{stat} = \rho_1 \cdot \left(\frac{f}{f_{min}}\right)^2 + \rho_2 \cdot \frac{f_{min}}{f} , \tag{2.1}$$

where the parameters ρ_1 and ρ_2 are derived from the E_{min} and E_{max} values [PLS01]. At a frequency increase, the dynamic part increases proportionally to the square of f while the static part decreases linearly due to the shorter computation time.

2.3.4.2 Performance Calculation

The performance evaluation of a set of tasks is based on the following equations for delay and for power consumption. First, the plain execution time T_e of a task, which runs on a device d at a cycle rate f, is calculated with

$$T_e = \vec{I_T} \cdot \vec{I_D} \cdot \frac{1}{f} , \tag{2.2}$$

by using the dot product between the instruction class mix vector $\vec{I_T}$ and the instruction class execution time vector $\vec{I_D}$. Taking all the tasks into account that run on device d, we define the total load L_D of device d as

$$L_D = \sum_{t \in \mathcal{D}} T_e(t) \cdot R(t) , \tag{2.3}$$

where $T_e(t)$ denotes the plain execution time of task t and $R(t)$ denotes its repetition frequency, which is identical to the application repetition frequency. By considering the device load, we define the effective execution time T_{ef} of some task as

$$T_{ef} = \frac{T_e}{1 - (L_D - T_e \cdot R)} . \tag{2.4}$$

We assume that on average a task is completed within T_{ef} [TCGK02]. The effective execution time is used by the exploration environment to verify the application's timing constraints. Finally, the power consumption P_D of a device d is calculated with

$$P_D = \begin{cases} P_s & \text{if } L_D = 0 , \\ (1 - L_D) \cdot P_i + L_D \cdot E \cdot f & \text{if } 0 < L_D \leq 1 , \\ \text{not valid} & \text{if } L_D > 1 . \end{cases} \tag{2.5}$$

We assume (a) that a device with zero workload switches to sleep mode, and (b) that a device that is not operating at full capacity switches to idle mode whenever possible. The term $E \cdot f$ represents the power consumption in execution mode.

Device	f_{min} [MHz]	f_{max} [MHz]	E_{max} [nJ]	Type
MSP430F13x	4.15	8	2	low-performance CPU
MSP430C33x[a]	1.65	3.8	4	low-performance CPU
PIC16LF87x-04	4	10	0.7	low-performance CPU
uPD78083, 8bit	–	5	0.55	low-performance CPU
StrongARM SA-1110	59	251	2.8	medium-performance CPU
XScale	150	1000	1.8	medium-performance CPU
SH-4	–	200	7.5	medium-performance CPU
AT91M40807	16	40	3.86	medium-performance CPU
TMS320C55xx (16bit)	–	200	1.7	integer DSP
TMS320VC-150	–	75	2.7	floating-point DSP
ADSP-2116x	–	100	1.9	floating-point DSP
PowerPC 440GP	–	400	7.5	desktop CPU
Ultra Sparc III	–	950	88.9	desktop CPU
TM5800	367	800	7.5	desktop CPU

[a] internal multiplier

Tab. 2: Some example computing devices

2.3.4.3 Device Set

The specification of the device set is based on a classification of the devices into five classes: low-power and low-performance CPUs (including PICs and micro controllers), low-power and medium-performance CPUs (system clock higher than 25 MHz), integer DSPs, floating-point DSPs, and desktop CPUs. For each class, representative devices have been selected and investigated as summarized in Table 2.

2.3.5 Resource Specification: Communication Channels

In the case of wearable systems, communication between the different modules plays an important role. In contrast to conventional distributed computer systems, there is a large variety of dynamically varying communication demands on one hand and a diverse set of possible communication media and protocols on the other. Therefore, we develop a corresponding unifying model that leads to the estimation of the relevant transmission parameters such as bandwidth, packet delay and power consumption for a shared communication channel with bursty transmission.

A communication channel c serves as interface between two modules that consume and generate data. The tasks running on the modules define the communication requirements for these channels. One objective in the design of the communication channel is to reduce the power consumption and hardware overhead for the transmission of data while still dealing with the dynamics of a wearable system. This goal can be achieved by means of two well known strategies: sharing of channels between different tasks that exchange data, and collecting data to enable a transmission in the form of bursts. First, we describe the parameters which characterize the tasks and communication channels.

2.3.5.1 Communication Channel Model and Specification Parameters

For the estimation of the communication properties we assume that each task periodically generates data. Each of the n periodic inputs k is characterized by sending l_k data units with a period τ_k. A maximal deadline δ_k is associated with every data sample of size l_k in order to specify when the transmission of the data sample needs to be completed (see also Fig. 7 left side).

All channels are assumed to be bidirectional and to support acknowledged error free data transfers. A channel type has a maximum data rate B_{max} and an end-to-end delay T_d of data packets. We distinguish between four operation states for each channel type: *transmitting*, where data is transmitted with the maximal data rate B_{max}; *receiving*, where data is received; *idle*, where the modules are still connected to each other but no data is transmitted; and *standby*, where the power consumption is low while the device can still be controlled.

A connection requires a transmitting and a receiving unit, introduced in the generic system model as communication channel interfaces. For the standby state, we thus have a power consumption of $2 \cdot P_s$. When data is transmitted/received, the power consumption is $P_a = P_{tx} + P_{rx}$. The transition from state standby to state transmitting/receiving takes time T_i with power consumption P_a. As long as the channel stays connected and is not transmitting/receiving data, the devices are in an idle state with a power consumption of $2 \cdot P_i$.

We consider two modes (Fig. 7): in continuous mode, the channel performs the sequence idle–transmitting/receiving–idle, and in burst mode we find standby–transmitting/receiving–standby. We suppose that the communication channel chooses the most power efficient mode that still satisfies the delay and bandwidth constraints.

2.3.5.2 Performance Calculation

The performance evaluation of the continuous and burst mode is based on the following equations for the power consumption and communication delay.

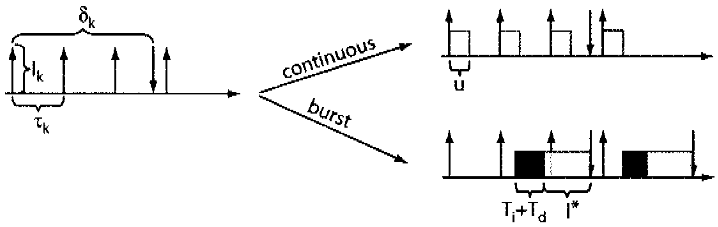


Fig. 7: Model of the communication demand of tasks (left) and continuous and burst modes (right).

Continuous mode

The worst case delay T_{wc} occurs if all channels submit their data at the same time, resulting in

$$T_{wc} = \frac{\sum_{k=0}^{n} l_k}{B_{max}} + T_d \ . \tag{2.6}$$

The channel power consumption is given by

$$P_C = u \cdot P_a + 2(1-u) \cdot P_i \ , \tag{2.7}$$

where $u = B_{max}^{-1} \sum_{k=1}^{n} l_k / \tau_k$ represents the channel utilization.

Burst mode

For modeling purposes, we assume periodic activity of a channel. We calculate the maximal period τ^* such that all delay constraints for the transmission are satisfied. In addition, we determine the maximal size l^* of a burst, i. e. the maximal amount of data to be transmitted. Based on this information, we calculate the power consumption. It can be seen that all deadlines are satisfied if the period of the channel is $\tau^* = \min_{1 \le k \le n}(\delta_k/2)$. We get the worst case delay for transmitted data packets

$$T_{wc} = \min_{1 \le k \le n}(\delta_k) + T_d \ . \tag{2.8}$$

Considering the case that all tasks start sending their data at the same time, we get the maximal burst size

$$l^* = \sum_{k=1}^{n} l_k \left\lceil \frac{\tau^*}{\tau_k} \right\rceil \ . \tag{2.9}$$

As we switch between two bursts from transmitting/receiving to standby and back to transmitting/receiving, we require that $l^*/B_{max} + T_i \le \tau^*$ for

Channel Type	P_{tx} [mW]	P_{rx} [mW]	P_i [mW]	P_s [mW]	T_i [ms]	$2T_d$ [ms]	B_{max} [Mbit/s]
RFM TR1000_PIC	39	16	12.8	7.8	100	50	0.115
Bluetooth P2P	151	150	71	17	950	155	0.768
Bluetooth P2M	204	188	134	17	890	128	0.768
Bluetooth PC-Card	490	425	160	160	1200	77	0.768
802.11a PC-Card	1416	1429	1406	129	1000	1	54
802.11a PC-Card PS[a]	1558	1525	119	93	1000	2	54
802.11b PC-Card	1115	1175	1035	225	1000	1	11
802.11b PC-Card PS[a]	390	450	235	225	1000	63	11
100base PC-Card	505	518	389	106	1000	1	100
UART Transceiver	125	125	0.99	0.0033	10	1	0.235
USB Bridge	149	149	3.3	3.3	100	1	12
Firewire Bridge	716	716	254	1.5	100	1	400
CAN Bus Controller	33	33	1.2	0.3	100	1	1
I2C Bus Controller	7.5	7.5	7.5	0.012	10	1	0.100

[a] power save mode

Tab. 3: Some example communication channel types

using burst mode. Finally, the power consumption is calculated by averaging the time intervals in which the communication channel is in its different states. We obtain

$$P_C = u \cdot P_a + \frac{T_i}{\tau^*} \cdot P_a + 2\left(1 - u - \frac{T_i}{\tau^*}\right) \cdot P_s \; . \tag{2.10}$$

2.3.5.3 Device Set

We consider wired and wireless channel types of the four classes: low-end wireless, high-end wireless, low-end wired, and high-end wired. For each class representative variants have been selected and investigated as summarized in Table 3.

Measurements of the power consumption in each operation mode have been performed for fully loaded channels. Furthermore, the startup time for initializing the device and setting up a channel T_i and the round trip time of a packet $2 \cdot T_d$ has been measured. For those channel types that were only available as a chip set, the data is derived from data sheets.

2.4 Exploration Environment

The generation of a candidate architecture involves two steps: (1) the combination of the generic architecture model with information from the problem specification, leading to the problem specific model, and (2) picking a particular configuration out of this search space.

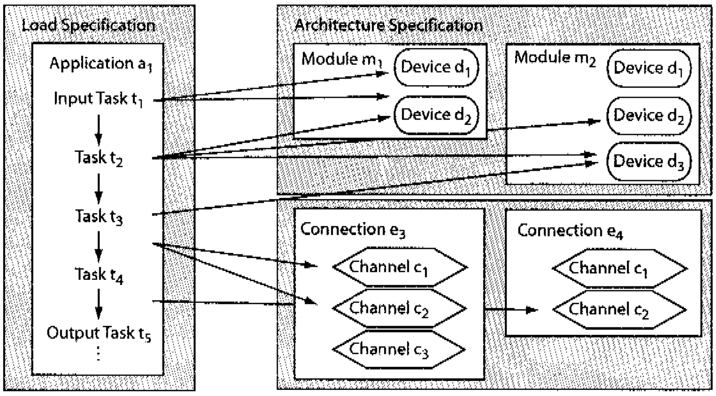


Fig. 8: The module resource set specifies the types and channels that a module can contain. The task resource set defines for each task on which modules and on which devices the task can run.

2.4.1 Problem Specific Model

The problem specific model provides a set of constraints on the system topology, the resources available within each computing module, and the assignment of tasks to modules.

- *System Topology:* The system topology determines which modules the system contains and how they can be interconnected. It specifies a set of modules $\mathcal{M}$ and an interconnection matrix I. For each pair of modules m_i, m_j the corresponding matrix element I_{ij} is set to 1 if a connection exists between them, otherwise it is set to 0.

- *Module Resource Set:* The module resource set specifies the types of devices and channels that a module can contain. For every module m, a subset of devices $\mathcal{D}^m \subseteq \mathcal{D}$ is defined that m can contain. Similarly, the selection of each connection is restricted to a subset of channels $C^{ij} \subseteq C$ from the set of available channels.

- *Task Resource Set:* The task resource set describes on which modules and on which devices a certain task can run. For each task t it specifies a subset of modules $\mathcal{M}^t \subseteq \mathcal{M}$ and a subset of devices $\mathcal{D}^t \subseteq \mathcal{D}$ on which t is allowed to be executed (see Fig. 8).

2.4.2 Architecture Representation

Picking a particular architecture out of the search space, which is defined by the problem specific model, is done through *allocation functions* and *bindings*. For every module m, the *device allocation* function $\alpha_D^{d,m} : \mathcal{D}^m \mapsto \{0, 1\}$ assigns to each device $d \in \mathcal{D}^m$ the value 1 (allocated) or 0 (not allocated). In a similar way the *channel allocation* function $\alpha_C^{ij} : C^{ij} \mapsto \{0, 1\}$ is defined for each connection between modules.

The assignment of a task to a particular device on a particular module is done by the *task-device binding.* Because of the correspondence between tasks and body locations given by the information flow, the task binding implicitly restricts each module to a subset of certain body locations: if task t is to be executed on module m, then module m can only be placed at a location $p \in \mathcal{P}^t$. Obviously, only such tasks can be bound to a module for which a valid location can be found. This means that the set of valid locations for a module is defined as the intersection of the locations of the corresponding tasks that are allocated to this module. The assignment of a module to one of the valid locations is performed by the *location binding.*

2.4.3 Architecture Evaluation

The basic approach is outlined in Fig. 2. The exploration environment performs an iterative search process. It maintains a set of wearable system architectures described by the allocation and binding functions. It iteratively adds new architectures by changing promising architectures and removes less promising architectures. The search process is conducted in five steps, where the architecture evaluation and architecture selection are described in some more detail.

Due to the modular nature of our methodology, the investigation of alternative algorithms and methods is possible in all of the five steps. However, the exploration of a huge search space demands for an efficient estimation of the quality of candidate architectures. In the current implementation, we apply first order models, which however, could be replaced by more sophisticated methods based on statistical modeling, simulation or traced-based approaches.

Based on the estimated execution cost and the performance of each architecture, the evaluation unit computes the values for the three optimization criteria.

2.4.3.1 Functionality

The functionality criterion is evaluated by means of the two application timing constraints R and D_{max}. While the repetition frequency is treated as a hard constraint, we regard the maximal latency as a soft constraint and allow it to slightly be exceeded. We use the accumulated differences of the estimated latencies D and the specified maximum latencies D_{max} to measure the functionality.

2.4.3.2 Battery Lifetime

As a quantitative measure of the battery lifetime, we use the average system power consumption. The following equations show how the power consumption is determined. The module power consumption P_{mod} is the accumulated power consumption of all allocated devices and channels for a specific module m in a scenario s. This sum is weighted with the element corresponding to location p of the power weight vector $\vec{w}_P$:

$$P^{m,s}_{mod} = \sum_{d \in \mathcal{D}^m} \alpha^d_D \cdot w_P \cdot P^{d,s}_D + \frac{1}{2} \sum_{c \in \mathcal{C}^m} \alpha^c_C \cdot w_P \cdot P^{c,s}_C \ , \tag{2.11}$$

where the device power consumption P_D and the channel power consumption P_C are calculated with (2.5), (2.7) and (2.10). The scenario power consumption P_{scen} of a specific scenario s is defined as the accumulated module power consumption P_{mod} of all modules m:

$$P^s_{scen} = \sum_{m \in \mathcal{M}} P^{m,s}_{mod} \ . \tag{2.12}$$

Finally, the system power consumption is defined as the accumulated scenario power consumption P_{scen} of all usage scenarios $s \in \mathcal{S}$, weighted with the scenario weights W_{scen}:

$$P_{sys} = \frac{\sum_{s \in \mathcal{S}} P^s_{scen} \cdot W^s_{scen}}{\sum_{s \in \mathcal{S}} W^s_{scen}} \ . \tag{2.13}$$

2.4.3.3 Wearability

The module wearability factor F_{mod} is the sum of the abstract wearability factors of all allocated resources for a specific module m at a given location p. The wearability vectors of computing devices $\vec{w}_D$ and communication channels $\vec{w}_C$ are weighted with the weight vectors corresponding to the location of the module:

$$F^m_{mod} = \sum_{d \in \mathcal{D}^m} \alpha^d_D \cdot \vec{W}^p \cdot \vec{w}_D + \sum_{c \in \mathcal{C}^m} \alpha^c_C \cdot \vec{W}^p \cdot \vec{w}_C \ . \tag{2.14}$$

The system wearability factor F_{sys} is defined by the sum of the module wearability factors F_{mod} of all modules m:

$$F_{sys} = \sum_{m \in \mathcal{M}} F^m_{mod} \ . \tag{2.15}$$

2.4.4 Architecture Selection

The main challenge with the exploration of the design space lies in its size, because of the combinatorial explosion of possibilities with an increasing number of devices, tasks and locations. There are several possibilities for exploring the design space, one of which is a branch-and-bound search algorithm where the problem is specified in the form of integer linear equations [Mic94]. However, in case of non-linear fitness functions and multi-objective criteria it is advantageous to use evolutionary search techniques [BTT98, ETZ00].

In addition, we are faced with a number of conflicting objectives trading the system wearability factor against power consumption. There are also conflicts that arise from the different usage scenarios, which are defined by sets of applications with associated maximum latencies and repetition frequencies. As a consequence, the binding of tasks to resource instances and the delay requirements may vary between scenarios. The goal of the selection unit is to determine implementations with Pareto-optimal fitness vectors. The architectures associated with Pareto-optimal fitness vectors represent the trade-offs in the wearable system design. In the current implementation we rely on an evolutionary optimization algorithm that has been successfully used for similar problems [TCGK02]. For the selection, we use the well known evolutionary multi-objective optimizer SPEA2 [ETZ00, ZLT01].

2.5 Exploration Results

In this section, we focus on the detailed discussion of a proof-of-concept case study. Although the considered system is not very complex, it represents an important class of wearable systems. We emphasize on the most important features of the simulation environment and provide first insights into the trade-offs involved in the design of wearable computing systems. We have also applied the concepts of the proposed design methodology to characterize the WearARM [LAT+01] and WearNET [LJS+02] demonstrator platforms. We explain the performance gain of the Pareto front compared to the WearARM/WearNET platform. To illustrate some further interesting points of investigation, we conclude the section with a brief discussion of an additional experiment: a system that includes a wireless connection to an external computing device.

2.5.1 Case Study Description

We consider a system in a mobile environment aimed at interactively displaying and recording JPEG images and having simple network access.

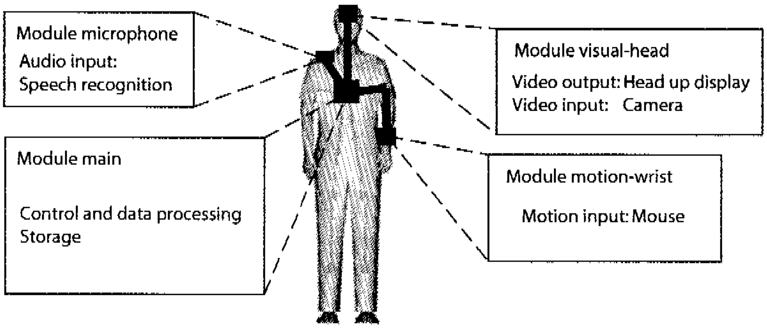


Fig. 9: Outline of the system investigated in the case study. Shown are the five locations with the assigned input/output tasks and the interconnection of the modules.

Fig. 9 outlines the example system. Information is displayed on a head up display unit, which also comprises an integrated camera for image acquisition. Speech commands via a microphone are accepted and a simple wrist-mounted motion sensor is used as mouse replacement. In addition, the motion sensor and the microphone are used for context monitoring. As has been shown in [LJS+02], the combination of these two sensors with appropriate background information on the user's whereabouts allow to derive complex information on his activity.

2.5.1.1 Usage Profile

We specify the workload of the system by five scenarios. Most of the time (70 percent) context monitoring and recognition tasks are running and evaluating the motion and sound signals. The remaining 30 percent of the time is equally partitioned into scenarios that combine the context monitoring applications with (a) an image display application, (b) an image recording application, (c) a speech recognition application, and (d) a set of network access related applications.

2.5.1.2 Information Flow and Physical Constraints

Each input/output task is assigned to one or several locations. In particular, the display output and the image recording input tasks are assigned to a location on the head (*visual-head*) to target a combined display/camera module worn in the glasses. The motion sensor input tasks are assigned to a location on the wrist (*motion-wrist*) and the audio input tasks to a location near the neck (*microphone*). A fourth location (*main*) is defined on the lower back for a central computing module, which connects all the peripherals.

The wearability weight matrix and the power weight vector are dic-

Designs:	A	B	C	WearARM/WearNET
Devices[a]				
main	XScale	XScale	XScale	SA-1110
visual-head	TMS320C55xx	TMS320C55xx	TMS320C55xx	TMS320VC-150
motion-wrist	-	MSP430F13x	MSP430F13x	MSP430C33x
microphone	-	-	-	-
Connections to module main[b]				
visual-head	Bluetooth P2P	Bluetooth P2P	CAN	USB
motion-wrist	RFM	RFM	I2C	UART
microphone	RFM	RFM	I2C	UART

[a] see Table 2 for full name and description.
[b] see Table 3 for full name and description.

Tab. 4: Example implementations of design points in Fig. 10

tated by the placement, with the location *visual-head* being highly sensitive to wearability and power factors and the location *main* being fairly insensitive to both. All connections can be either wired or wireless, but we punish wired connections with increased wearability weights. In particular for the connections *visual-head*↔*main* and *motion-wrist*↔*main*, we consider wired connections expensive, because wires running between different body parts are rather disturbing. For the two locations *visual-head* and *motion-wrist*, wired channel interfaces are punished by a factor of 5 and the location *microphone* by a factor of 3 compared to the location *main*.

2.5.1.3 Resource Specification

For the exploration of the case study system we use the hardware devices described in Sections 2.3.4 and 2.3.5 and listed in Tables 2 and 3.

2.5.2 Results

Fig. 10 shows the final population of a design space exploration run after 500 generations with a population size of 100 architectures. The optimization run took less than 20 minutes on a SunBlade 1000. Each dot in Fig. 10 represents a Pareto-optimal system architecture including the set of allocated devices in the modules, the choice of channels for the module connections, and the binding of tasks to devices for each scenario. Table 4 lists the device and channel allocation of three selected design points and our WearARM/WearNET system to give an example of the trade-offs involved in the wearable system design. Fig. 11 illustrates the example architecture B.

The allocation of the channels is a dominating factor. The architectures B and C have the same CPU resources, but a different communication channel selection. Power and wearability differ by a factor of 3. In general, we can say that in the lower right region there are exclusively

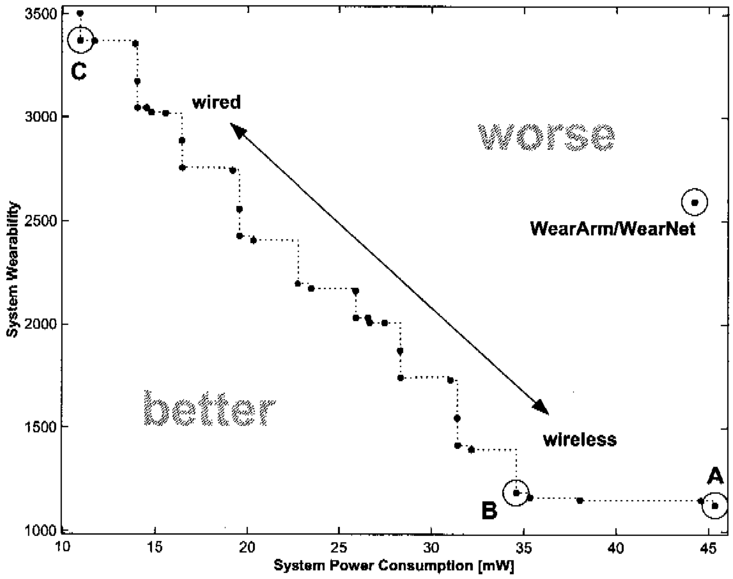


Fig. 10: Final population of a design space exploration run. The dotted line represents the Pareto front with the Pareto-optimal design points. All points on the upper right side of the Pareto front are dominated by at least one Pareto point. The figure also shows three selected architectures (A, B, C) and the WearARM/WearNET system for comparison.

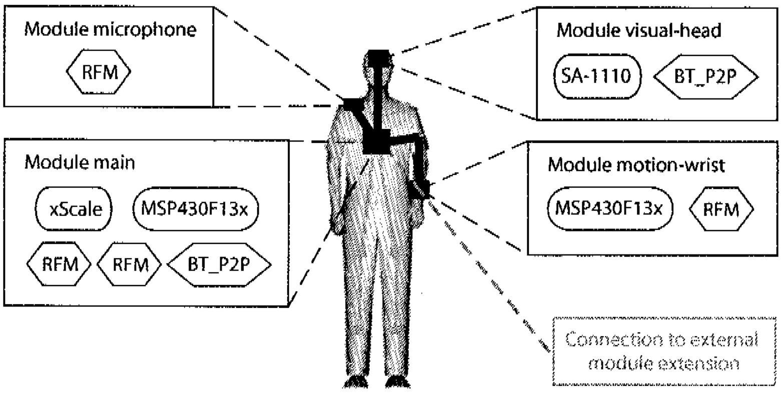


Fig. 11: Example design point B. A connection to an external module extension is illustrated (see Section 2.5.3).

solutions with wireless connections. Moving to the upper left region we find solutions that increasingly use wired connections, which are more power efficient but less wearable.

We have applied our design methodology to characterize our WearARM/WearNET system. The fitness values are about a factor of 2 away from the Pareto front. However, we can explain the improvements required to reach the Pareto front. The StrongARM SA-1110 needs to be exchanged with the latest Intel ARM CPU (XScale), which was not yet available during system design. The same applies to the newer DSP in the visual module. To reach the Pareto point C, we need to switch from the standard computer connections (e. g. RS232) to more power-efficient ones (e. g. I2C bus).

We observe that the smallest implementation—architecture A—is not completely centralized. The reason for this is the frame buffer, which at least requires a simple processor to be assigned to the *visual-head* module. Given the necessity of such a device, it may be advantageous to allocate a more powerful device, which is capable of executing additional tasks such as JPEG encoding and decoding. Indeed, the exploration has shown to evolve to exactly this solution.

2.5.3 External Module Extension

For computationally intensive tasks, wearable computers often use external compute servers to which data is sent for processing via a wireless connection [TPB98, KKS01]. In our methodology, such an external server can be modeled as a module that requires a wireless connection. Since wearability and power consumption of an external server have no impact on the wearable, we set the wearability and power weights to zero. This means that any resources on the server are 'free', except for the communication costs.

As an additional experiment, we have chosen an image recognition application that optionally compresses and decompresses the image to reduce the costs of outsourcing. This setup has allowed us to perform a series of simulations varying the computational intensity, i. e. the image size, of the recognition task. Surprisingly, the external module was only used in those cases where no mobile processing device was able to perform the computation. In all other cases architectures without a link to the external module were chosen. The costs of maintaining a high-speed wireless link outweighed any computational savings.

The results of the exploration are different, if we force an external wireless channel such as WLAN to be part of the system (which is true for most networked systems). In this case, outsourcing some tasks to the external device is indeed more power efficient. The question as to which tasks are to be outsourced depends strongly on the computation to communication ratio and the type of the wireless connection.

2.5.4 System Robustness

The resulting architectures of a design space exploration run depend on a large number of parameters for the usage profile, the hardware resources and the architecture evaluation. In order to check the robustness of our implementation against small changes in the parameter set, we have applied a sensitivity analysis.

We have assigned an uncertainty range to each parameter class. We also performed several simulations with a parameter set that has been randomly modified within the specified uncertainty range. Fig. 12 shows the result. The solid line represents the simulation result with the unmodified parameter set, whereas the dotted lines denote the simulations with an uncertainty of up to 30 percent. The encircled design points represent one sample architecture. All simulations within this uncertainty range have come up with this characteristic architecture. However, if we further increase the uncertainty of some parameters such as the device power consumption, we see substantial differences in the resulting architectures of the simulations. This analysis provides a sense of the robustness of the system and of the implications of changes in the parameter set.

2.6 Summary

Wearable computing systems are intelligent, environment aware systems unobtrusively embedded into the mobile environment of the human body. The design of such heterogeneous and distributed systems needs to address a variety of conflicting criteria. These include the ability to efficiently execute dynamic workloads, the necessity of placing sensors and IO modules at different locations on the user's body, and stringent limits on size, weight and battery capacity.

The performance estimation of such systems considering the given constraints is difficult. We developed an estimation method that is based on the analytical models of (a) the envisioned usage profile, (b) the physical constraints, (c) the information flow requirements and (d) the deployed computing and communication hardware resources. Estimation algorithms are developed yielding performance, power consumption, cost and system wearability measures. Embedded in an automatic and multi-objective design space exploration environment that evolves a set of Pareto-optimal wearable architectures, these algorithms provide a methodology for reliable, quantitative analysis and systematic design of wearable systems.

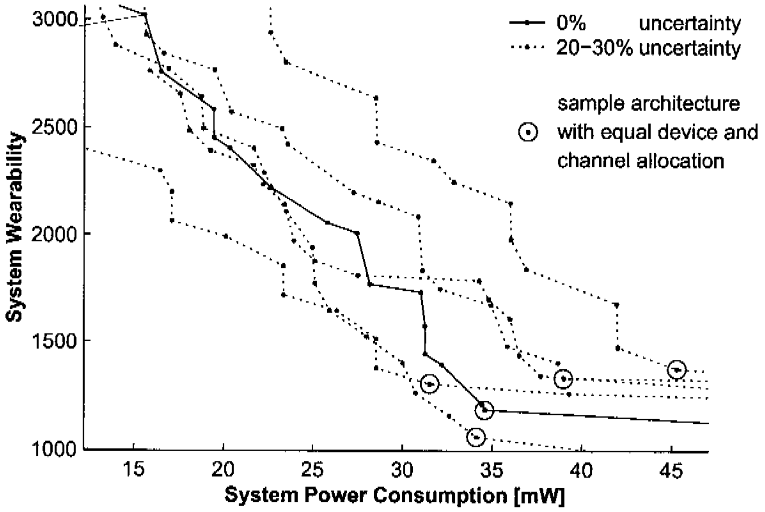


Fig. 12: Solution set of a sensitivity analysis. Each line is the Pareto-front of an individual simulation run where we randomly modified the ressource specification (Table 2 and 3). For every parameter, a random relative uncertainty between 20 and 30 percent has been added. The encircled design points represent one sample architecture. All simulations within this uncertainty range have come up with this characteristic architecture.

3

FPGA Emulation and Virtualization of Hardware Tasks

Field-programmable gate array (FPGA) emulation is traditionally conducted to validate application-specific integrated circuits (ASICs). Since finding and correcting errors in hard-wired ASICs is an enormous effort and very expensive, the circuits are prototyped on FPGAs and validated there. FPGAs are reconfigurable. Therefore changes in the circuit can be made with minimal effort.

In recent years, FPGAs have become larger and more efficient. As a consequence they are also employed as custom computing machines. The flexibility of such systems can be improved by dynamically changing the circuit during operation. The validation of such run-time reconfigurable systems is a difficult task. We propose the following method to address this problem: The application running on the reconfigurable system is split into so-called hardware tasks. We investigate a formal coordination language that defines the interaction and interface between the tasks. With this abstraction, it is possible to validate individual tasks separately and to check the interaction between tasks, since this information is given explicitly by the coordination language.

In this chapter we present the emulation and virtualized execution of hardware tasks on a FPGA. Section 3.1 describes the architecture and related work of a reconfigurable embedded node. Section 3.2 describes a hybrid emulation environment, that efficiently executes applications consisting of multiple hard- or software tasks. The application is virtualized in the sense that the description of the task interaction is given in a formal language, which is interpreted. Since the information of the task coordination is explicit, validation techniques such as simulation and formal methods can directly be applied. In Section 3.3 we propose a design-flow for partial reconfiguration. Finally, we present two different prototype implementations of our concepts in Section 3.4 and summarize the chapter in Section 3.5.

3.1 REnode – A Reconfigurable Embedded Node

An important requirement of distributed wireless embedded systems such as sensor networks or wearable systems is flexibility. This can be shown on the basis of three properties:

Multi-mode performance: Sensor networks and wearable systems are multi-mode systems. They require a fixed, baseline amount of performance for running control tasks that do not show high computational demands. Occasionally, e.g. upon detection of an event, bursts of computation-intensive tasks have to be executed [PEW+03].

Low energy: Energy awareness is essential for battery-operated nodes. It comprises several measures. First, we have to run the high performance bursts on computing elements with high energy efficiency. Second, to provide the baseline performance, computing elements optimized for low-power must be used. Third, a dynamic power manager should force the components into their power-down modes or even shut them off completely.

Dynamic adaptation: An embedded node has to handle dynamic situations. First, as it is not feasible to design a new dedicated hardware platform for every application, the platform should provide the flexibility to adapt to different application requirements. Second, even for a given application, the system has to adapt to changes in the environment, e.g. operating a sensor network in a dense deployment could be very different to a sparse deployment.

Flexibility demands can be addressed by using programmable general-purpose computing units, such as microcontrollers or CPUs. However, high-performance and low-energy demands ask for specialized hardware. We address this challenge by proposing an architecture that comprises reconfigurable hardware: *the reconfigurable embedded node.*

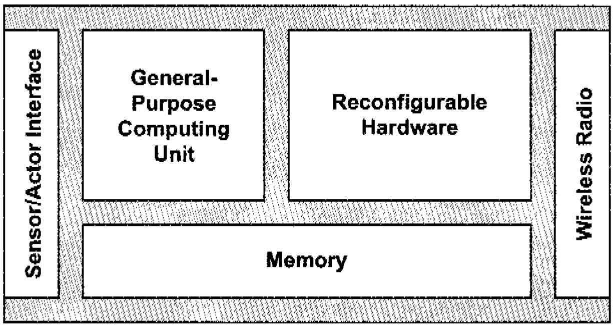


Fig. 13: REnode Hardware Architecture

3.1.1 Architecture Overview

Figure 13 shows the hardware architecture of the reconfigurable embedded node (REnode). It consists of a CPU, a reconfigurable hardware unit, memory, a set of I/O interfaces to sensors and actors, and a wireless interface for communication with other nodes.

The CPU handles control and other tasks that need low to medium processing power. The reconfigurable hardware unit executes tasks with high computation demands, but can also run communication protocol functions to relieve the CPU. The reconfigurable hardware is further used for interfacing to external sensors and actors. Both the CPU and the reconfigurable hardware unit have power save modes.

3.1.2 Embedded Reconfigurable Hardware

The predominant reconfigurable hardware device today is the field-programmable gate array (FPGA). FPGAs consist of an array of complex logic blocks (CLBs), or an array of slices containing two or more CLBs, routing channels to interconnect the logic blocks and surrounding input/output (I/O) blocks. SRAM-based FPGAs use SRAM cells to control the functionality of the circuit and can be reprogrammed arbitrarily often by downloading a stream of configuration data to the device.

While early FPGA generations were quite limited in their capacities, today's devices feature millions of gates of programmable logic and, additionally dedicated hardware blocks such as fast embedded memories and fixed-point multipliers. To interface to external components, FPGAs are compliant to a number of high speed I/O standards. Current FPGAs have sufficient resources to implement rather complex circuits such as cryptography algorithms, audio, image, and video processing functions, networking interfaces, and complete CPU cores.

A currently popular trend is to combine FPGAs with CPUs to form hybrid computing systems, or so-called configurable systems on a chip

(CSoC). Examples of such hybrid systems are Triscend's 8051-based E5 [Tri01], Xilinx's PowerPC-based Virtex-II Pro [Xil02], and Atmels AVR-based FPSlic [Atm03].

With respect to performance, power consumption and flexibility, reconfigurable hardware is positioned between processors and dedicated hardware (ASICs). Several case studies have shown that FPGAs achieve higher throughput and are more energy-efficient than processors, provided that the application matches well the spatial structures of FPGAs and includes a sufficient amount of parallelism.

Mencer et al. [MMF98] and Abnous et al. [ASI$^+$98] compared different implementations of signal processing algorithms on embedded RISC and DSP processors and on an FPGA. Their result is that the FPGAs achieve the highest performance. The energy efficiency relates the performance to the power consumption. For all applications, the FPGAs achieved a better energy efficiency than the embedded RISC processor. The DSPs outperformed the FPGAs in energy efficiency for FIR and IIR filters, because these filters perfectly match the DSP architectures. Stitt et al. [SGVV02] implemented a set of benchmarks on the Triscent E5 hybrid CPU and measured an average energy saving of 71% by moving application kernels to the FPGA instead of running the applications solely on the CPU.

3.1.3 Dynamic and Partial FPGA Reconfiguration

FPGAs are configured either statically or dynamically. In the case of static configuration, the FPGA loads the configuration data typically from an external non-volatile memory at system startup. The configuration does not change during the system's runtime. In the case of dynamic configuration, an external host processor writes the configuration data to the FPGA. This allows for changing the FPGA configuration on demand. Some FPGAs offer an advanced configuration mode, partial reconfiguration, which allows for the configuration of parts of the device at runtime. A partial reconfigurable FPGA is comparable to a multi-processor architecture: it can execute different processes truly in parallel and it is run-time programmable.

FPGA manufactures have realized partial reconfiguration differently. In this thesis, we restrict ourself to the most prominent partial reconfigurable devices, the Virtex Family from Xilinx.

The Virtex FPGA reconfiguration is organized in frames. Frames are the basic units of reconfiguration and determine the settings of all FPGA resources in the vertical dimension (see Figure 14).

An FPGA is reconfigured by sending a configuration bitstream to the device. The configuration bitstream contains a collection of frames. While the bitstream of a full reconfiguration contains all frames, the bitstream

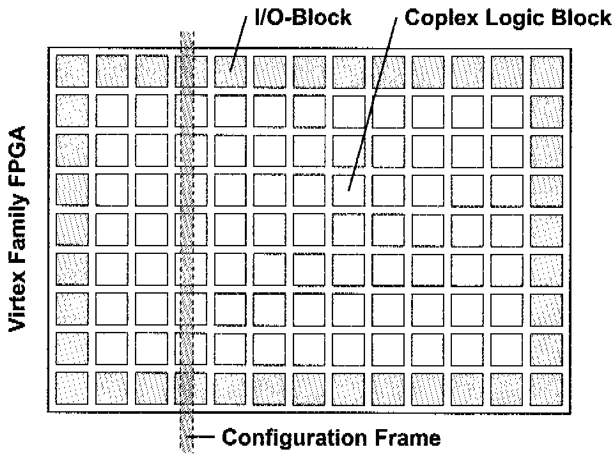


Fig. 14: Configuration Architecture of the Virtex Family FPGAs

of a partial reconfiguration only contains selected frames. This bitstream is referred to as a *partial bitstream.*

3.1.4 Hardware Tasks

A central concept in the REnode architecture is the hardware task. A hardware task is a part of an application that runs temporally on the reconfigurable hardware. It is usually stored as a partial bitstream. An important property of a hardware task is its footprint, which is defined by the area and the shape of the occupied logic. Since the FPGA has a fixed amount of logic cells, the number of hardware tasks, which can run in parallel, is limited.

A hardware task can be of different complexity and size. For instance, an *Adder-Task* that simply adds two values only uses a few CLBs, while a reconfigurable processor-task is much larger and could easily occupy half of all logic blocks in the FPGA.

Figure 15 shows a possible floorplan of the reconfigurable hardware in a REnode. Besides the dynamic reconfigurable hardware tasks and the static parts of the application, an interconnect logic is required that transports data to and from the tasks.

Similar to a personal computer, a REnode needs an operating system that manages the available system resources, i. e. the CPU, the reconfigurable hardware unit, the memory, and the I/O. Defining an operating system for dynamic reconfigurable hardware is a challenging task and is a new research field. The elementary questions here are the following:

- How is an application defined and how is it executed?
- How is a task defined and how is it scheduled, loaded and executed on the reconfigurable device?

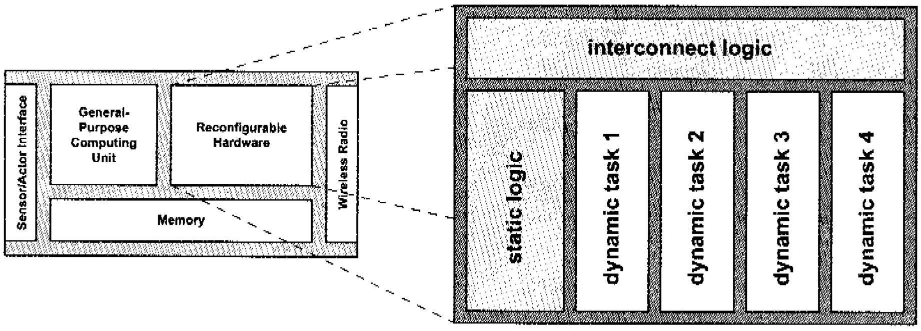


Fig. 15: Structural floorplan of the reconfigurable hardware: Example with four reconfigurable tasks.

- What are the interfaces between two tasks and between a task and the I/O?

Defining a partially reconfigurable multi-tasking system is a difficult task. Furthermore, the implementation of a working prototype is very challenging. One reason is that in industry, reconfigurable devices are still used mainly statically. As a consequence, the design- and tool support for the new partial reconfiguration technique are very limited.

For the REnode architecture we further need to address the flexibility demands of a networked embedded device. Our approach is the virtualized execution of process networks, which is described in the following section.

3.2 Virtualized Execution of Process Networks

In this section, we investigate the virtualized execution of dynamic tasks on reconfigurable embedded nodes. A virtual machine run-time system is introduced, which efficiently executes streaming applications. We first highlight the advantages of an interpreted coordination language for design and validation. We propose the use of the popular process network model and describe its implementation in hardware. It is then shown how a run-time system can execute arbitrary large process networks by using dynamic reconfiguration, a memory manager and a scheduler.

An application for an FPGA can be virtualized on different levels. Table 3.2 shows two hierarchical levels of abstraction. Following a model-based design approach, an application is explicitly specified by a directed graph where the nodes are tasks, which represent computations and the arcs represent communication. The tasks are arbitrary subprograms and are specified in a conventional programming language such as C or VHDL, but the interaction between tasks is defined by a precise semantics. We call the language defining this interaction the *coordination language*. Examples of coordination languages are *process networks*, *synchronous dataflow (SDF)* graphs or *petri-nets*. In model-based design, those or related models of computation are frequently used to specify applications.

Abstraction Level	**Languages**	**Elements**
Coordination & Communication	Process Networks, SDF, Petri-Nets	Network Graphs, Tasks
Task Functionality	VHDL, C	Instructions, Gates

Tab. 5: Two Levels of Abstraction for an Application

A language can be either interpreted or compiled. Our approach is to virtualize and interpret only the coordination language while the task functionality is compiled. Therefore, an application is specified in two parts. The first one describes the coordination, i.e. the communication and the synchronization between tasks. The second part is the set of pre-compiled tasks, e.g. in the form of partial configuration bitstreams. Our target architecture is a reconfigurable embedded node, including a CPU and reconfigurable logic. We present a novel use of an FPGA as a computing element for streaming-based applications, with the investigation of dynamic reconfigurable tasks on a virtualized run-time system and an interpreted coordination language.

3.2.1 Virtualized Execution - Related Work

Today, it is commonly accepted to specify signal processing and streaming applications in a coordination language. In [KDR01], a compiler is presented, which transforms algorithms written in Matlab into a process network. A static FPGA implementation (without reconfiguration) of such a process network specification is given in [ZSKD03]. This approach uses two compilation steps to generate VHDL code which is then synthesized. Thus, both the task coordination and the functionality are compiled into one piece of program.

A virtualization of an entire FPGA application has been presented in [HSE$^+$00] and [LK03]. Both of these approaches define an interpretable hardware byte code with limited functionality. In [HSE$^+$00], the virtual byte code also includes the routing information of the FPGA interconnect. The disadvantage of these approaches is the overhead and the performance loss, arisen from the virtualization on the level of gates and interconnects. In contrast, we compile the task functionality, using only a virtualized coordination language.

The use of precompiled tasks and an interpreted coordination language has been presented in [HK02] for control-flow applications on a CPU based embedded system. For reconfigurable hardware systems, this is a new research field. Related to this is the work which has been done on *operating system services* for reconfigurable hardware. This includes hardware multitasking [Bre96]; device partitioning, placement and routing [WK02]; task preemption and scheduling [SLM00] [BD01] [WP03b] [WP03a]; and hardware/software relocatable tasks [MNC$^+$03].

The SCORE [CCH$^+$00] compute and execution model is closely related to the work presented in this section. However, in contrast to SCORE, we present a run-time system that is compliant with today's available FPGA devices. In addition we present a prototype implementation and performance results that can be compared to other similar system architectures.

The mentioned related work has contributed to our goal of developing and implementing a dynamic reconfigurable run-time system for the reconfigurable embedded node. While one group uses a specific coordination language as a model of computation, but only in a static scenario without reconfiguration, another group is working toward a general operating system, which does not investigate a specific model for the optimized execution of streaming-based applications. In contrast, we present a system which investigates both the dynamic reconfigurability and a specific coordination language for streaming-based applications.

3.2.2 Task-Level Virtualization

We propose the virtualization of an application on the level of tasks. To interpret and execute an application, given as an abstract description in a coordination language and a set of precompiled tasks, a run-time system is needed. The run-time system is a virtual machine, since it abstracts the underlying hardware and it provides the execution of tasks as a service to the application programmer. It hides the exact implementation of the service but guarantees an execution with the semantics of the coordination language. With this paradigm, system designers and application programmers can benefit from a number of advantages.

Device dependency:

The aspect of the application that is specified by the coordination language is device independent. Only the individual tasks have to be ported to a new architecture. The tasks are dependent on the device family, but not directly on the device size. To execute a task on a larger FPGA, only minor modifications are needed, which can be integrated in an automated design-flow.

Component-based design:

Using a coordination language allows the application programmer to easily build new algorithms by a new composition of existing tasks. Since the interface between the coordination language and the task functionality is precisely defined by the model, the programmers can work concurrently on the implementation of tasks and the definition of applications. Furthermore, tasks can be validated separately, which is less difficult than validating a complex application consisting of multiple tasks. To cope with the increasing complexity and size of applications, developers can build a library of validated tasks which they can use for several applications.

Performance:

FPGAs achieve high performance by parallel operation. Our approach supports parallelism in two ways. First, a task can use the parallelism to speed up a local computation. Secondly, the run-time system can load several tasks onto the FPGA, which operate concurrently. However, there is a restriction on the parallelism. If there is not enough area on the FPGA for all the tasks, the virtual machine uses time-multiplexing with run-time reconfiguration. As a consequence the performance of the system will be dependent on the size of the device and the configuration time of the tasks. However, with time-multiplexing, even a small system with a virtual machine can execute —with less performance— an arbitrarily large process network. Another important factor influencing the performance is the tools being used for compilation. With architecture-dependent tasks, we can use the efficient synthesis-, place- and route tools from the manufactures.

Validation:

The coordination definition of an application carries information about the tasks and their interconnection. Since we do not compile the coordination language, this information is retained. With this information we can analyze the performance of the system with simulation or formal methods.

3.2.3 Task Coordination Language

There are many candidate coordination languages and models which could be used for an embedded reconfigurable node. Some coordination languages are better suited to model control-flow oriented applications, and others are intended for data-flow applications. As an example,

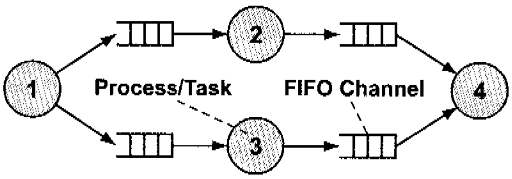


Fig. 16: Example of a Process Network

we have chosen the Process Networks model to use as a coordination language. Process networks are a popular coordination language for streaming-based multimedia and signal-processing applications. An application is modeled as a collection of concurrent processes, which communicate through unidirectional FIFO channels (see Figure 16). Each of the processes performs computation on its private state space. The computation is interleaved with communication actions that read data from input channels and write data to output channels.

The process network model fits our requirement for a task coordination language, since it has simple semantics and makes task-level parallelism and communication explicit. It is further a very flexible model. Compared to more restrictive models such as synchronous dataflow (SDF) graphs, it is also harder to analyze analytically. However, we can regard the process network domain as a superset, which includes some dataflow domains (dynamic dataflow, boolean dataflow and SDF) as subdomains [Par95]. We can therefore increase the expressive power of our coordination language for better analyzability, by including additional information and restrictions. One example of such an addition is given in Section 3.2.5.

In [Kah74], Kahn has defined a precise semantics of a process network, which is today known as the Kahn Process Network (KPN) model. The communication actions in a Kahn process are sequential, i.e. the process can access only one channel at a time. In the KPN model, the processes can not test an input channel on the availability of data. If a process tries to read from an empty channel it is suspended until there is enough data in the channel to complete the read action. As a consequence, KPNs are deterministic, i.e. the history of data produced on the communication channels is determined by the given input data and does not depend on the execution order of the processes. Our run-time system supports the execution of KPNs, but the processes can also optionally use a mechanism to allow undeterministic applications as proposed in [dKSvdW$^+$00]. The undeterministic case is further discussed in section 3.2.5.2.

The properties and limitations of the KPN model and the requirements of its execution have been studied by Parks [Par95] and Geilen et al. [GB03].

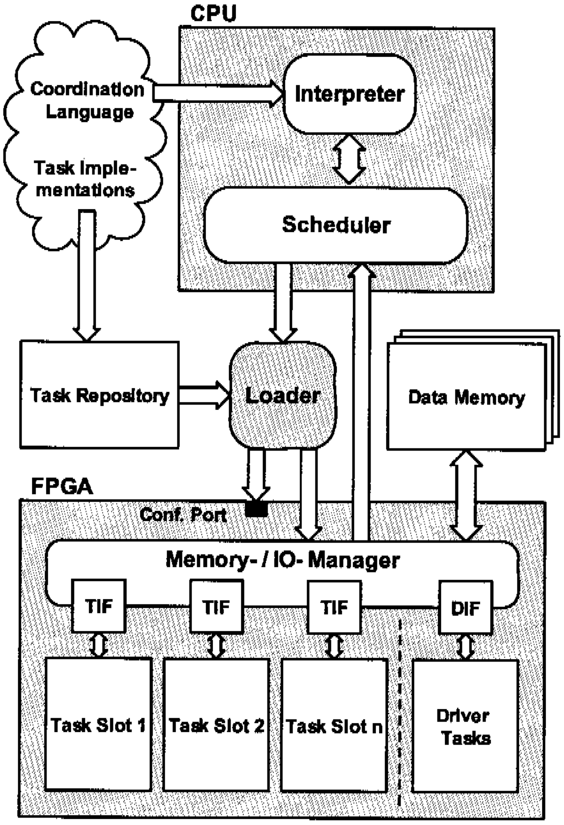


Fig. 17: The virtual machine run-time system of the reconfigurable embedded node.

3.2.4 REnode Run-Time System

An overview of the run-time system is depicted in Figure 17. The main unit is a partial reconfigurable FPGA from the Xilinx Virtex Family. The computation is done in the task slots, which are continuously reconfigured with tasks from the task repository by the loader. The tasks communicate over the task interface with the memory- and I/O-manager. The memory- and I/O-manager acts as a crossbar for the data memory and sends events to the scheduler. Based on this events, the scheduler and the interpreter decide which task to load next.

3.2.4.1 Implementation of Tasks

The tasks correspond to the processes in the process network model. They have a number of communication ports which are virtually connected to the according FIFO channel. The ports are enumerated and are declared either as an input- or output port. In order to load a task as a partial reconfigurable core, we need a precisely defined, static task interface and protocol. We have joined the different ports of a task to a single interface with a data-, control-, and a port select bus. This implementation implies that the task can only use one port simultaneously, which reflects the property of the KPN model. In our current implementation, the application programmer is responsible for the sequentialization of the communication. Conceptually, this could also be done by an automatic wrapper function.

A problem that has to be addressed is the context of the tasks. A task usually has an internal state and variables. In streaming-based multimedia applications a task usually runs for a long time —e.g. when playing a movie— until it terminates. When the scheduler preempts a task, its context therefore needs to be saved and restored after its next instantiation. One possible solution is to let the task write its own context to an additional FIFO channel, from which it can also read back after configuration. However, the burden of context restoring should rather be on the run-time system than on the application programmer. Alternatively, the loader could read back the complete task over the configuration port as proposed in [WP03b]. The configuration port is a possible bottleneck and the time for the readback is as high as for the configuration. Therefore we use an internal context save mechanism. We apply an automated post-synthesis step which connects all used registers of the task into multiple shift registers, only by inserting multiplexers and signals. In the task interface of the run-time system, there is a controller which connects to this shift-registers. In this way, the context can be shifted out of the task word-wise and be stored in memory. The same structure is used to load the context back.

3.2.4.2 Memory Manager

The tasks have a simple interface to the memory manager. A task selects one of its ports and performs a read- or a write action. The FIFO channels are implemented as circular buffers and the data is stored in the data memory of the run-time system (see Figure 17). The memory manager computes the addresses for the data memory and handles the communication. For a flexible management of the FIFO channels, the memory manager uses internally a virtual address space. Allocated regions of the virtual address space are mapped to the physical address space.

When a new application has to be executed on the run-time system, the interpreter allocates for each FIFO channel a region in the virtual address space. In the FIFO state structure all information of the FIFOs are stored, including the virtual base address, the size and the read- and write positions. The memory manager computes the physical address by computing the virtual address first and then by translating it to a physical address. The first step is done locally for every task slot, whereas the

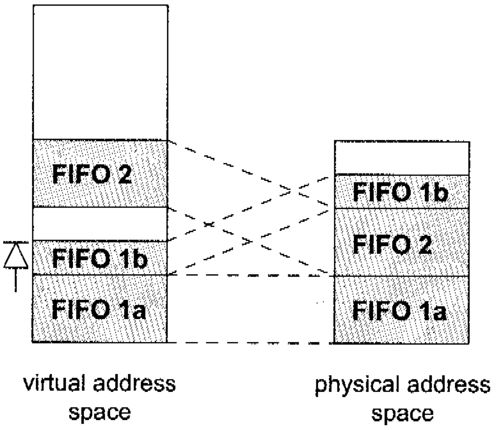


Fig. 18: Allocation of FIFO channels in virtual- and physical address space

second step is done system wide. The virtual address is computed by a FIFO control logic. If a new task is loaded into a task slot, the FIFO control logic of that task slot is configured with the FIFO state of those FIFOs, which are used by the task. With this information and the network information the FIFO control logic can efficiently compute the virtual addresses and update internally the read- and write positions. Before a new task is loaded, the read- and write positions of the old task are written back to the FIFO state structure.

The function, that converts the virtual addresses to the physical addresses can be configured such that fragments of a region in the virtual address space can be stored at different locations (see Figure 18). This allows for example to increase the capacity of a channel at run-time, which is important for the scheduler, as it is explained later in section 3.2.4.5.

To increase the memory bandwidth, the data memory can consist of multiple physical devices. We also allow to use the on-chip dual-ported block-memory. Each of the physical memory devices is attached to a memory crossbar and has an arbiter. E.g. if two tasks try to access one physical memory, the arbiter selects one of them to be deferred. We discuss the exploration of different memory configurations further in the performance results (section 3.2.6.2).

With this memory manager architecture, most of the communication actions are possible to complete within one or two clock cycles. But this efficient implementation makes it hard to scale to an increased number of slots. One reason of this is the inherently bad scaling of components such as the memory crossbar or the memory arbiter. However, we believe that with today's available devices, implementations with up to 16 slots are feasible.

3.2.4.3 Driver Tasks

Driver tasks are the connection from the process network to the environment. Driver tasks are typically connected to I/O pins of the FPGA to handle external devices, such as streaming input- or output devices. This handling usually requires the driver tasks to be present continuously in order to meet the timing requirements e.g. of a communication protocol. As a consequence a driver task can not be reconfigured and scheduled by the run-time system. We have addressed this problem by creating a static slot, which includes all the driver tasks. The driver interface to the memory manager is different from the task interface. It can be adapted to the needs of the drivers. A driver tasks can for example be statically connected to one port of a on-chip blockram, whereas the second port is available for the tasks.

3.2.4.4 Partial Reconfiguration

The loader receives commands from the scheduler. A load command basically consists of the number of the task to load and the number of the target slot. Tasks are stored in the task repository as partial bitstreams. The loader modifies the header information inside the bitstreams to set the target of the reconfiguration to the appropriate slot. The modified bitstreams can then be sent to the configuration port of the FPGA. The time for partial reconfiguration linearly depends on the size of the bitstream. The configuration port can be a possible bottleneck in the system, since only one slot can be reconfigured at a time.

In a system on a chip (SoC) realization of the reconfigurable embedded node, the loader is a separated control logic connected to the CPU, the reconfigurable hardware and the task repository. However, the loader could also run on the CPU, or even on the FPGA itself. The Xilinx VirtexII FPGAs allow *self configuration*, where the logic inside can use an internal configuration access port (ICAP).

3.2.4.5 Scheduler

Conceptually, all tasks in the process network run in parallel. In our run-time system, this task-level parallelism is reduced to the number of task slots in the FPGA. The scheduling strategy defines the execution order of the tasks and how long the tasks occupy a slot. The choice of a scheduling strategy depends on the task- and the resource model. In [WP03b], different scheduling strategies have been studied for a similar resource model, but a different task model. To execute process networks, a dynamic scheduling strategy is needed.

Compared to a CPU architecture, the context switch process (saving states, reconfiguring and restoring states) takes rather long and can not be neglected. In order to reduce the number of context switches, a task should run as long as possible before being preempted. We apply a data driven scheduling [Par95] strategy, where tasks are activated on the availability of data. We restrict our description of the scheduler here to the execution of deterministic KPNs. The undeterministic case is explained later as an extension of the model in Section 3.2.5.2.

A task is running until it reads from an empty channel or writes to a full channel (blocking reads/writes). Since the read- and write actions are sequential, a task can only block on one of its ports. The memory manager has to provide the scheduler with two events (see Table 6).

Event Type	*Arguments*
fifo-full/empty	slot num, fifo num
fifo-ready	fifo num

Tab. 6: Events for the basic scheduler

The *fifo-full/empty* event is generated, when a read to an empty FIFO or a write to a full FIFO is detected. The *slot num* argument defines the slot number of the FIFO control logic, which has handled the communication and the *fifo num* argument is the identifier of the FIFO channel. The *fifo-ready* event only needs the *fifo num* argument. It is generated when a particular channel has been full or empty and now becomes ready, because a task has performed a read- or write action on that channel.

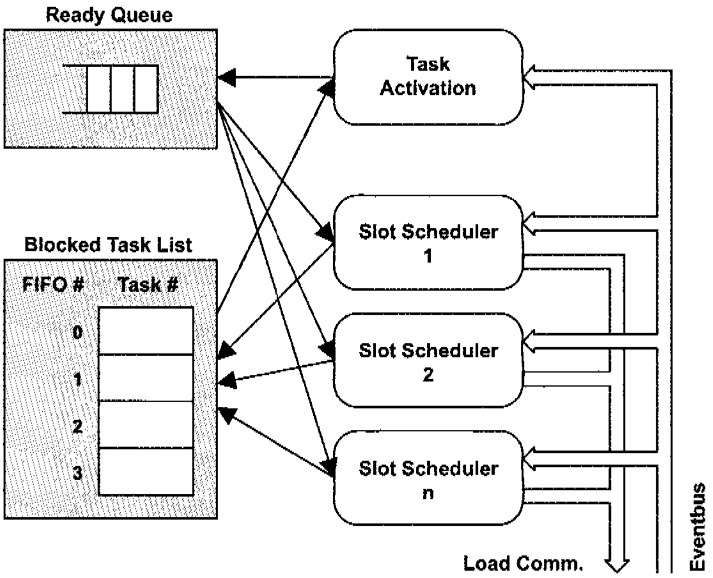


Fig. 19: Principle of the basic scheduler

The scheduler consists of a *task activation* thread and a number of *slot scheduler* threads (one per slot). In the KPN model, tasks are either active or blocked. An active task is either currently running in one of the task slots, or it waits in the *ready queue* to be loaded. If a task blocks, the scheduler thread of the corresponding slot adds the task to the *blocked task list* (see Algorithm 1). When the *task activation* thread receives a *fifo-ready* event, it clears the entry of that FIFO in the *blocked task* list and adds the corresponding task to the *ready queue* (Algorithm 2).

Algorithm 1 Slot Scheduler

```
loop
  GetTaskFromReadyQueue(&TaskNr) // wait if queue is empty
  SendLoadCommand(TaskNr, ThisSlotNr)
  repeat
    WaitFor_FifoFullEmptyEvent(&SlotNr, &FifoNr)
  until SlotNr = ThisSlotNr
  AddTaskToBlockedTaskList(FifoNr, TaskNr)
end loop
```

Algorithm 2 Task Activation

```
loop
  WaitFor_FifoReadyEvent(&FifoNr)
  TaskNr = GetTaskFromBlockedTaskList(FifoNr)
  ClearBlockedTaskListEntry(FifoNr)
  AddTaskToReadyQueue(TaskNr)
end loop
```

Since the process network model does not specify a ratio of the amount of data consumed and produced on the ports of a task, such as in the SDF model, the execution of certain process networks might end in an artificial deadlock. An artificial deadlock occurs, if all tasks are blocked because of the limited FIFO channel capacity. This problem can be addressed by integrating a *deadlock-resolution* algorithm into the scheduler. In his thesis, Parks [Par95] proposes a simple algorithm to resolve artificial deadlocks in KPNs. It increases the capacity of the smallest full FIFO channel in the system when a deadlock occurs.

The scheduler can detect an artificial deadlock, by inspecting the blocked task list. If all tasks of an application are in the blocked task list, the scheduler starts resolving the deadlock with Parks algorithm. To allow the capacity of the channels to grow over time, they are allocated in the virtual address space, such that they are not adjacent. The capacity of a channel can thus be increased by allocating a new fragment (see Figure 18), by updating the FIFO state structure and by reestablishing the order of the data in the circular address space.

3.2.5 Restrictions and Extensions to the Coordination Language

In order to further increase the flexibility and the performance of our system, we propose a restriction and an extension of the classic KPN model.

3.2.5.1 SDF Tasks

A KPN task is not restricted to have a constant input or output rate. In the DSP domain though, there are many tasks with have exactly this property. Examples of such tasks are filters, constant bitrate decoders and fast Fourier transformation, just to mention a few. We call these tasks *SDF tasks*, since they can be specified with the restricted semantics of the SDF model. However, the run-time system, as it has been described so far, can not yet use this property to efficiently allocate the FIFO channels with an initial capacity. Consider the task B in Figure 20, which has to read two samples from the input in order to produce on sample on the output.

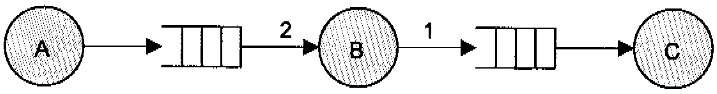


Fig. 20: Example of an SDF Task

Without the knowledge of the input and output rates of task B, both FIFO channels will be allocated with the same capacity. However, unless task A and task B run in parallel, the second channel will only be used to the half of its capacity. To allocate the second channel half as large as the first channel would be more efficient. We have addressed this problem by allowing optionally the specification of the input- and output rates of tasks. The capacity of a channel, which is adjacent to a SDF task, is calculated with the additional information of the input- and output rates, whereas the other channels are allocated with the same size.

3.2.5.2 Undeterministic Communication

To model undeterministic communication, the task interface is extended with an additional communication action. Besides a read- and a write action, a task can issue a *select* action. The formalisms of this action is given in [dKSvdW$^+$00]. In the select action, the task selects a number of ports, that should be guarded. If there is data available on at least one of the selected input ports, or free space available on at least one of the selected output ports, the select action immediately completes and returns the task one of the ports, which has fulfilled the above condition.

Otherwise, the task is suspended and the select action is forwarded to the scheduler as an event.

The algorithms of the scheduler are modified such that on a select event, the task is added in the blocking list in the entry of all the channels that are connected to a selected port. If one of the channels becomes ready, the task activation thread reactivates the task and clears all other entries of that task in the blocking task list.

3.2.6 Emulation based Validation

In order to explore the design space of different slot- and memory configurations, we have implemented a parameterizable run-time system emulator. While the simulations in previous work are usually based on a randomly generated task set, the emulator is able to execute and profile real applications.

The emulator is implemented in VHDL. We use the ModelSim VHDL simulator to execute and profile our application. The task slots and the partial reconfiguration are emulated by a crossbar switch, which can connect every task of the application with a specific task interface in the memory manager. Thus, as input for the emulator, we can use exactly the same tasks as later in the the synthesis for the prototype implementation (see Section 3.4.2).

The emulator is parameterizable. The following parameters can be set to explore different solutions in the design space:

- the number of slots,
- the number of physical memory devices,
- for each memory device, the number of simultaneous access (1 for normal external memories, 2 for dual-ported memories),
- the capacity of the memory devices, and
- the time from an event to a load command in the scheduler.

The input of the emulator is an application with the following components:

- The compiled tasks (stored as entities in a library),
- The coordination definition of the application (in the task coordination language) including
  - the interconnection of the tasks,
  - the identification of driver tasks,
  - the properties of the tasks, including (a) the size of the partial bitstream, (b) the number of context words (flip-flops that store the state of a task) and optionally (c) the input and output rate of a task.

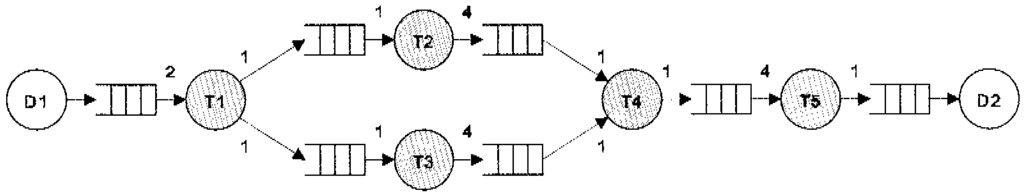


Fig. 21: Process network of the sample application.

3.2.6.1 Sample Application

We have modeled a sample application with our coordination language and implemented the tasks in synthesizable VHDL code. We have chosen a streaming-multimedia application. The process network graph is depicted in Figure 21.

The application is a transcoder with a stereo-to-mono mixer in the ADPCM compressed format. It consists of seven tasks, including two driver tasks. The driver task *D1* receives a joined stereo ADPCM stream and puts the received data periodically on the first FIFO channel. The task *T1* separates the left and the right channel. The tasks *T2* and *T3* are both ADPCM decoders, which decode the ADPCM streams to a raw 16-bit PCM format. Both decoders use the same partial bitstream for configuration, but they have an own instance with an own state. Task *T4* is the actual mixer. It computes the average of the left and the right channel. Task *T5* encodes the raw PCM stream back to the ADPCM format and the driver task *D2* sends the stream over its connected interface.

Table 7 shows two properties of the tasks used in the sample application. The context bits store the state of a task.

Task	*Size in CLBs*	*context bits*
T1	44	32
T2/T3	266	96
T4	44	48
T5	340	96

Tab. 7: Sample Application Task Properties

3.2.6.2 Performance Results

In order to conduct a number of experiments and to obtain quantitative data, we have executed the sample application with our run-time system emulator, described in the previous section. The first experiment shows the effect of the total memory capacity and the number of slots on the performance and latency of the system (Figure 22).

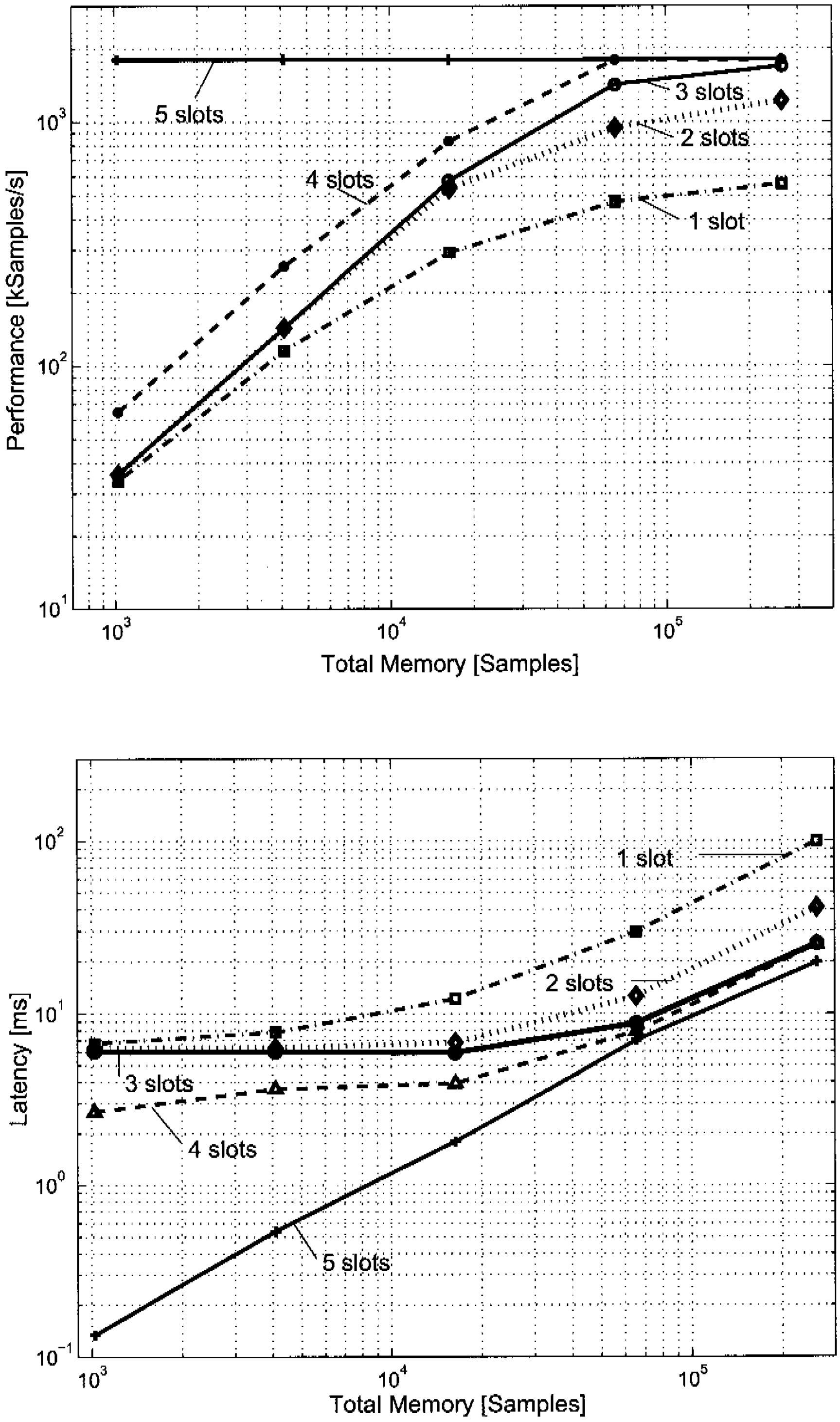


Fig. 22: Performance and latency of the sample application in the run-time system.

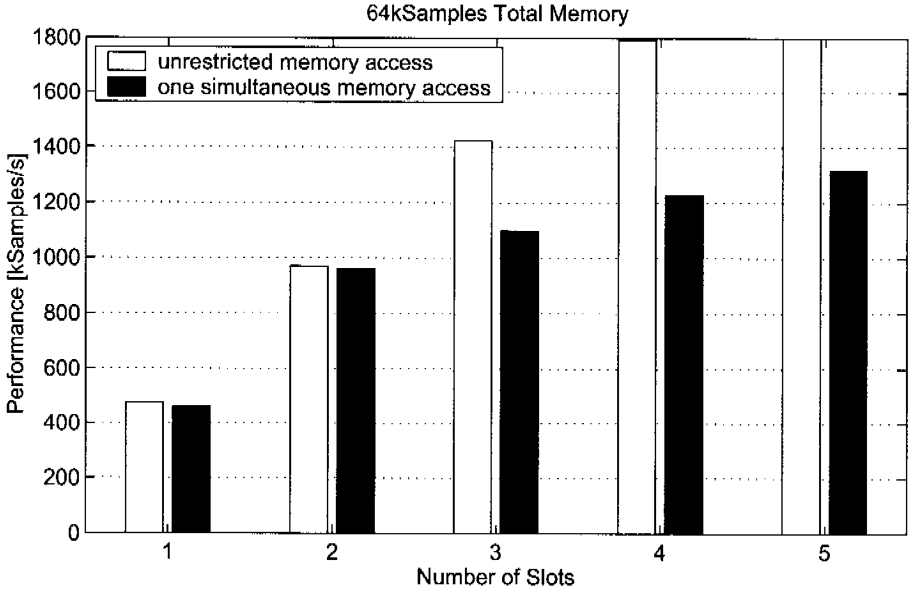


Fig. 23: Performance with restricted access to the data memory.

We have measured the performance for systems with one slot, where the tasks are sequentially configured, up to five slots, where every task runs in its own slot. Note that the sample application has two driver tasks, which are stationary and therefore not loaded into a slot. In this experiment we did not restrict the number of simultaneous memory accesses.

The performance of the five-slot system is not dependent on the total amount of memory as the other systems, since after the first instantiation, no further reconfiguration is required. This system actually corresponds to a static solution. Its performance of about 1.8 megasamples/s is defined by the performance of the ADPCM encoder task, which is the slowest task of the application. The overhead in terms of execution time of the other systems is due to the reconfiguration and not to the virtualization. With little memory, the tasks fill or empty the FIFO channels with little capacity so fast, that the task slots stay most of the time in the reconfiguration state or the state which waits for the access to the configuration port. The performance, but also the latency increases if more data memory is used.

So far, we did not restrict the access to the data memory. The maximal number of possible simultaneous memory accesses is equal to the number of slots plus the number of driver tasks. However, the access is restricted by the memory configuration. In Figure 23 we show the effect of restricting the access to the data memory to one simultaneous access such as in our prototype. The experiment is done with a total memory capacity of $64K$ samples.

In the Figure, we can see that the static five-slot solution is faster by a factor of two, compared to the two-slot system, with unrestricted memory access. If there is only one single-ported memory device such as in our prototype, this factor decreases to 1.37.

3.3 Partial Reconfiguration Design Flow

While the last section described the execution of the tasks, we present in this section a design flow for the generation of optimized tasks as partial bitstreams.

When the first partially reconfigurable Virtex FPGAs were released, developers only had the possibility to directly manipulate configuration bits inside a bitstream. The basic concept is described in the next section.

3.3.1 Direct Bitstream Manipulation

Standard design implementation tools for FPGAs generate full configuration bitstreams. The structure of Virtex bitstreams is partly open to the public, which allows to directly manipulate such bitstreams.

A rather simple manipulation is to change the contents of storage elements such as lookup tables (LUTs) and BlockRAMs in a bitstream. Such a technique has been used, for example, to customize logic functions at download time for instance-specific SAT solvers [LSW+01]. By extracting the relevant frames from a full configuration, partial bitstreams are generated. In combination with LUT modifications, runtime customization of FPGA cores becomes feasible. This technique could be used, for example, to dynamically change coefficients of a digital filter. In principle, this technique can also be used to generate partial configurations for reconfigurable tasks. Figure 24 shows two full configurations, each containing a static core and a dynamic task. The set of frames containing the dynamic task can be extracted directly from the full bitstream to form a partial configuration. At runtime, the partial configuration is loaded on demand.

The main advantage of direct bitstream manipulation is that it bases on full configuration bitstreams that can come from arbitrary standard FPGA synthesis and design implementation tools. These tools are laid out to optimize circuit qualities, such as speed and area. Further, when the partial configurations modify only the contents of LUTs and BlockRAMs, the direct bitstream manipulation is quite simple and efficiently implemented.

Direct bitstream manipulation shows two limitations. First, for more complex designs the low-level manipulation of bits in a monolithic bit-

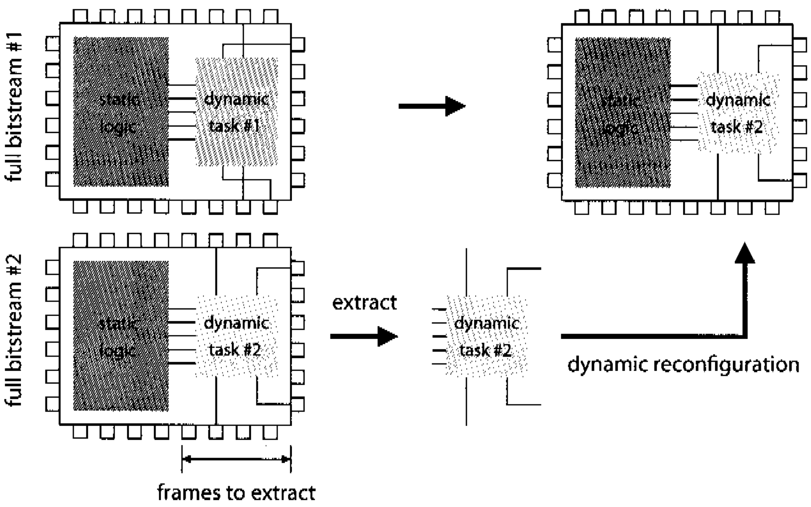


Fig. 24: Extracting a partial bitstream from a full bitstream.

stream becomes extremely tedious. Second, as the routing cannot be changed, different reconfigurable tasks must occupy exactly the same subarea of the FPGA, and the interface between the static logic and the tasks must be bound to a fixed location. Moreover, it must be ensured that the routes for the static logic do not run through the partially reconfigured area and vice versa. The design and implementation tools that were available at the time when this work was done, did not allow to pose location constraints on routing resources. Therefore constraining the routing became more or less a trial-and-error process, involving manual intervention.

3.3.2 Bitstream Generation and Manipulation Tools

In the meantime, a number of tools have been developed that help researchers to access resources inside a bitstream. The Xilinx JBits SDK [GLS00] and the Partial Bitfile Transformer PARBIT [HL01] are two examples.

JBits for example provides access to most of the Virtex resources through a Java class library. All Virtex resources can be instantiated and configured. At any time, JBits can save changes to the design as partial bitstreams. Up to now, JBits supports structural circuit design only, but it enables hierarchical designs by grouping subcircuits into modules or cores. JBits also includes an automatic router, which can dynamically route and unroute connections. In the design flow described in [SJR01], JBits manipulates designs, given as EDIF netlists, that have been fully mapped and placed by synthesis tools. JBits adds the routing and generates configuration bitstreams.

These tools offer two advantages over direct bitstream manipulation. First, they introduce a higher level of abstraction as they operate on CLBs, routes, etc., rather than on raw bits in a bitstream. This feature also opens up future dynamic applications, where tasks can be relocated and connections can be re-routed online. Second, these tools include versatile functions for full and partial bitstream manipulation.

3.3.3 Combining High-Quality Synthesis with Bitstream Manipulation

To get the best features from both worlds, standard design flows and the bitstream manipulation tools can be combined. We present a design flow that synthesizes the static and dynamic tasks with standard design implementation tools, and generates the partial configuration bitstreams with JBits. Our approach uses the bitstream manipulation tool to merge the cores. The locations for the reconfigurable tasks and the interface between static and reconfigurable tasks are bound to fixed locations.

Our design flow allows to generate an initial full configuration and a number of subsequent partial configurations. We discuss the details of the design flow on the example of a static core and one reconfigurable task. The design flow relies on two techniques: the *virtual socket*, a fixed-location interface between the static logic and the dynamic task, and *feed-through components* to constrain routing.

The virtual socket is a component that provides fixed locations for a set of pre-defined signals. All signals from the static core or the I/O pins to the task and vice versa are routed through this interface. The only exceptions are the global nets for clock and set/reset signals. Because the interface is static, new tasks can be developed without having access to the static core design.

The overall FPGA area is divided into two non-overlapping parts, one part for the static core and the other one for the reconfigurable tasks. The generation of the initial full configuration involves following steps (shown in Figure 25a):

1. Create an initial design which consists of the static core and a reconfigurable task component. Any synthesis tool or core generator can be used to derive high-quality designs.
2. Insert the virtual socket, a predefined interface component. Connect the static core and reconfigurable task via the virtual socket.
3. Run FPGA back-end tools with constraints on the locations of the static core, the reconfigurable task, and the virtual socket. This generates the initial full bitstream.

The tool flow respects the characteristics of the Virtex frame-based

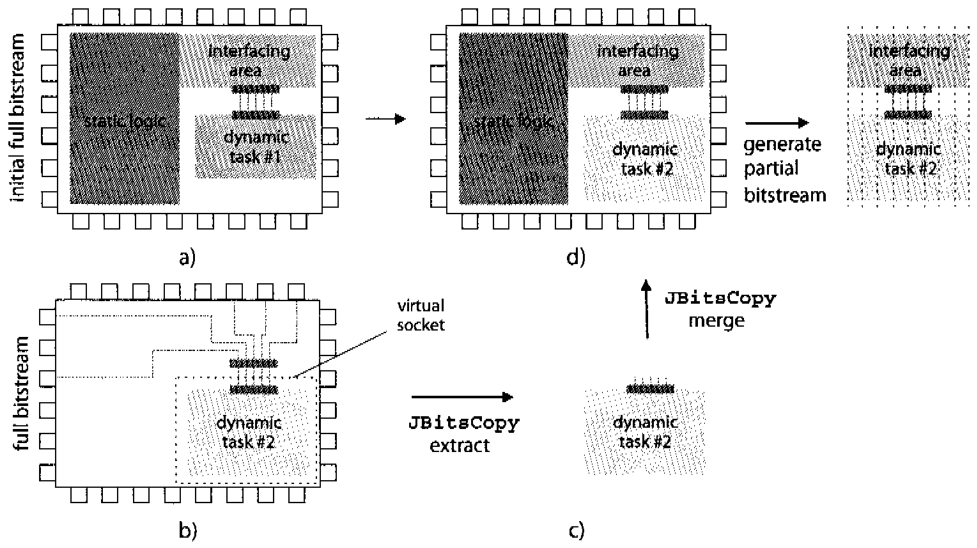


Fig. 25: Generation of initial full and partial bitstreams.

configuration mechanism. The part of the static logic that is located in the same (vertical) frames as the dynamic task area, denoted as *interfacing area* in Figure 25, contains only stateless resources. A partial bitstream for a reconfigurable task is generated as follows:

1. Create the task design with any front-end tool.
2. Connect the task to the predefined virtual socket. The static logic bound signals of the virtual socket are connected to unused I/O pins to prevent the optimizer from removing the socket component.
3. Run FPGA back-end tools with constraints on the locations of the task and the virtual socket. This generates a full bitstream (shown in Figure 25b).
4. Use a bitstream manipulation tool (e.g. JBits) to i) extract the task from the full bitstream (shown in Figure 25c), and ii) merge the task with the initial full bitstream (shown in Figure 25d). By this, the initial reconfigurable area is overwritten with the new task. The new task fits seamless into the initial design, provided the location constraints for the task and the virtual socket have been respected. The bitstream manipulation tool is then used to generate a partial configuration bitstream that reflects the reconfigurable area.

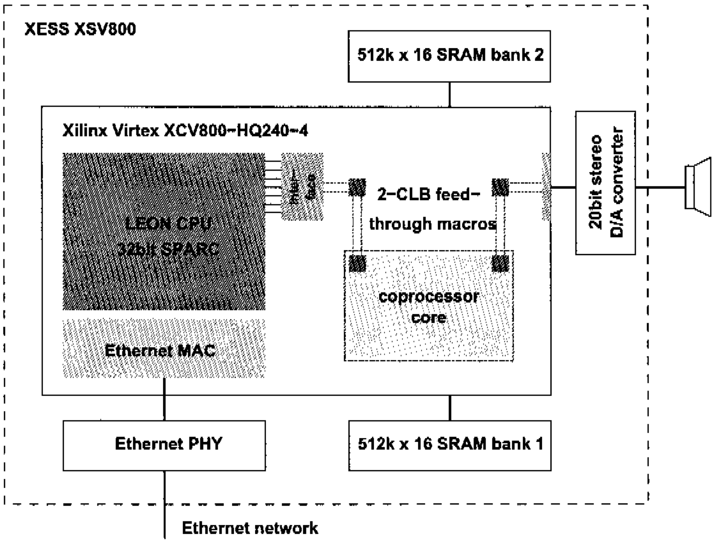


Fig. 26: Block diagram of the audio streaming prototype.

3.4 Case Studies and Prototypes

3.4.1 Reconfigurable Coprocessor

As case study and proof of concept for the reconfigurable design flow presented in Section 3.3, we have implemented a complete and fully operational audio decoding application. The prototype consists of a minimal embedded computer based on a general-purpose CPU core, memories, network interface, and several coprocessors for hardware-accelerated playback of audio streams.

The CPU receives UDP packets containing encoded audio data from a network via an Ethernet interface. The CPU unpacks the audio data and sends it to the audio coprocessor's input FIFO via the virtual socket. The coprocessor decodes the audio stream and sends the raw audio data to the on-board digital-to-analog converter. Different formats for the encoding of audio data require different coprocessors. Depending on the audio format currently used, the audio decoders are dynamically configured into the prototype. The technical details of the prototype are:

- *Prototyping Platform*
  The prototype has been implemented on a XESS XCV800 board which consists of a Xilinx Virtex XCV800-4 FPGA and a multitude of I/O interfaces. A block diagram of the prototyping board is given in Figure 26.

The cores were designed in VHDL, synthesized and implemented using Synopsys FPGA Express 3.6.0 and Xilinx Foundation 4.1i tools, respectively.

- *CPU core*
  The soft CPU core is the SPARC V8 compatible 32bit LEON CPU. The CPU was configured with 2kB separated data- and instruction-cache (implemented in internal BlockRAM), a 256 byte internal boot-ROM (implemented in internal distributed RAM) and an external 32bit memory interface. The CPU core requires 3865 Virtex slices (7730 CLBs) which amounts to 41% of the XCV800's logic resources, and 14 BlockRAMs which equals 50% of the memory resources. Without any optimization, the CPU runs at 25 MHz. Applications for the LEON core run on top of the RTEMS real-time operating system and are compiled using the GNU C based LECCS cross-compiler kit.

- *Coprocessor cores*
  We have implemented two audio decoding coprocessors, a PCM and an Intel/DVI compliant ADPCM decoder. The ADPCM core uses 430 Virtex slices (860 CLBs), or 4.5% of the XCV800 resources; the PCM decoder fits into 35 slices (70 CLBs), or 0.4% of the resources.

We envision the prototype application as a typical scenario for future embedded networked systems that load hardware functions on demand. The current limitations of this prototype are that the reconfiguration has to be initiated by the user and that the partial configurations are loaded onto the FPGA from a host computer via a configuration port.

This prototype shows the feasibility of dynamic reconfigurable hardware tasks. In the next section we present a more advanced prototype of the reconfigurable embedded node that includes a run-time system.

3.4.2 PDA-FPGA: Reconfigurable Embedded Machine

We have build a prototype of the reconfigurable embedded node as a proof of concept. The prototype is capable to efficiently execute process networks as described in Section 3.2. The experimental system is a small mobile assembly of an IPAQ PDA and a custom FPGA module (see Figure 27). The FPGA module is connected via the *expansion sleeve interface* to the PDA as memory-mapped I/O. The FPGA module consists of a 200k Gate Xilinx SpartanII FPGA, 256 kByte SRAM, 8 MBit Flash memory and a 384 Macrocell Xilinx Coolrunner CPLD. Although the FPGA has only one task slot, we have implemented all the key components presented in this paper, including the task interface, the FIFO control logic and the memory manager. The loader is mapped to the CPLD, which reads configuration bitstreams stored in the flash memory and configures the FPGA over the

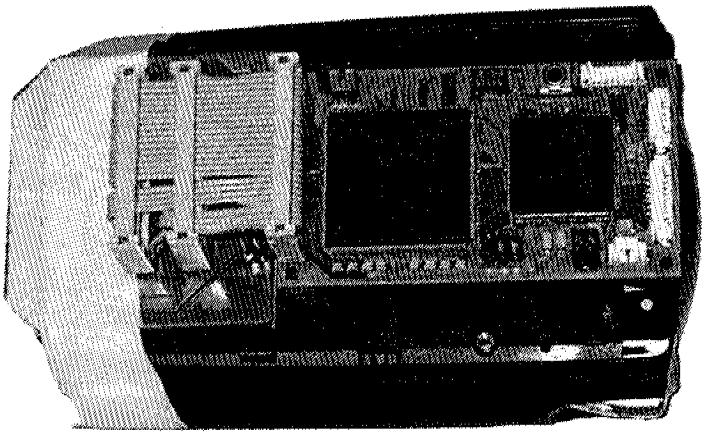


Fig. 27: Prototype of the reconfigurable embedded node with an IPAQ and a custom FPGA module.

SelectMap configuration port. The scheduler and the interpreter resides in the CPU of the PDA. The memory manager in the FPGA is connected to an external interrupt in order to send events to the CPU.

The size of the unoptimized memory manager is 800 slices (1600 CLBs), which is about one third of the SpartanII200 FPGA. This value depends on the FIFO control logic, which currently uses distributed RAM cells to locally store and update the state of the FIFOs. The above given slice count is based on a FIFO control logic, which can handle the communication of tasks with up to eight ports.

In our current implementations, the slots have a fixed size. Although the tasks have a different slice count, the size of the partial configuration bitstream of the tasks is rather defined by the width of a task slot. We use therefore for all tasks the same bitstream size of $16.7K$ bytes, which correspond to 235 slices or ten percent of a full bitstream of the SpartanII200 FPGA. With a configuration clock of 50 MHz, the partial reconfiguration time is about $330\mu s$.

The power consumption of the entire FPGA module is 500 mW on average. This reflects the fact that the current Virtex family FPGAs are optimized for performance and not for low power operation.

3.5 Discussion and Summary

In this chapter, we presented the virtualized execution of hardware tasks on a reconfigurable embedded node. We investigated partial reconfiguration where hardware tasks can be loaded on run-time. Using FPGAs as dynamic reconfigurable computing machines is a relative new field and the validation of such systems is challenging. We have addressed this problem the following way: We defined hardware tasks with a specified interface. We presented a design flow for the generation and reconfiguration of the tasks. We introduced a formal coordination language based on the process network model that specifies the interaction between the tasks. Our proposed run-time environment interprets the coordination language and executes the hardware tasks efficiently.

The virtualized execution has a number of advantages for design and validation: The interface and the interaction between the tasks are precisely defined by the model of the coordination language. This allows for component-based design and validation. Tasks can be implemented and tested separately with less effort. Furthermore, since the task interaction is interpreted and not compiled, this information is retained and can be used for validation.

We have built and evaluated two prototypes implementations of a reconfigurable embedded node. The first prototype features a dynamic reconfigurable coprocessor that is loaded as a hardware task on demand. With the second prototype we demonstrate the run-time environment that interprets the coordination language and virtually executes applications that are composed of hardware tasks. We have further built an emulator for the second prototype with which we can evaluate the performance with different configurations.

With the prototype implementations we have shown the feasibility of the REnode architecture. It is capable to execute arbitrarily large applications. With an interpreted coordination language we can easily define and deploy new applications which addresses the flexibility demand of a networked embedded system.

4

Distributed Algorithms

The previous chapters addressed validation strategies for some specific distributed embedded applications such as wearable or reconfigurable systems. In the remaining chapters, we discuss the validation of wireless sensor networks. We believe that the validation of WSNs is more challenging than most other systems. This can be illustrated by comparing different applications and systems by the measures of *data*- and *system uncertainty* (see Figure 28).

The *data uncertainty*-axis denotes the complexity of the stimuli of the systems. Validation effort, scalability and especially realism directly depend on how well we can understand and model the stimuli. For instance, it is much easier to validate systems that only have text as input than systems that are tightly embedded into a complex environment such as robotics systems. The *system uncertainty*-axis denotes the complexity of the application. The more to the right a system is, the less deterministic is its operation. Thus, the insight into a system, the visibility which helps for the validation, is worse on the right side. An application that runs on distributed multiple processors is much harder to debug than single processor- or single threaded applications.

Wireless sensor networks are in the worst position with respect to these two measures. It is very hard to model the stimuli such as the topology of a deployment, the wireless channel, the node mobility, the sensor inputs, etc. Furthermore, the algorithms are distributed across a large number of nodes that do not have a common time basis. Thus, the system uncertainty is high.

Validating distributed algorithms for wireless sensor networks is very

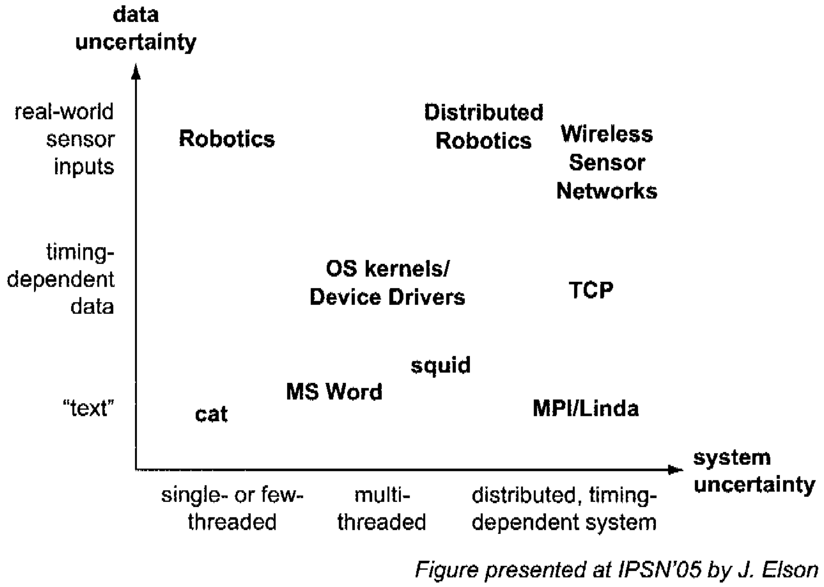


Figure presented at IPSN'05 by J. Elson

Fig. 28: Comparison of applications and systems by the measures of data- and system uncertainty.

challenging. Because of the data- and system uncertainty, it is important that algorithms are validated not only by simulation, but also with methods that provide higher realism such as test-bed measurements. Recent research in WSNs has produced a myriad of distributed algorithms, covering different topics such as MAC layers, topology control, clustering, routing, etc. However, if we look at the literature, we see that only few algorithms have been implemented and tested with realistic conditions. Most algorithm have only been validated using simulation with relative simplistic models.

The feedback from validation is required for designing practical algorithms. The feedback provided by simulation alone is not enough as the results depend on models that tend to be overly simplistic. We show that through implementation, we achieve a high quality feedback that gives us the required information for adapting the algorithms to the peculiarities of the physical environment. This supports the thesis that distributed algorithms for wireless sensor networks must be validated in a realistic environment.

In this chapter, we discuss distributed algorithms for the topology control problem. We start with a simple tree-building algorithm which we validated on real sensor nodes. With the experience gained from this algorithm we have choosen a more advanced, existing algorithm which has interesting properties, but has been validated only by simulation. We analyzed the assumptions made by the algorithm and the simulation and check whether these assumptions are realistic. In order to make it practical, we enhanced the algorithm with new heuristics, which we validated using formal analysis and simulation. Finally, we implemented the enhanced algorithm and validated it on real sensor nodes in an indoor test scenario.

4.1 Practical Topology Control

4.1.1 The Topology Control Problem

Many applications for wireless sensor/actuator networks (WSNs) require that all nodes are connected in a common network. Consider for example a sensor network for environmental monitoring, where the nodes regularly report measurement data to a central base station. In other applications, possibly with distributed real-time control such as tracking applications or wireless alarm systems, the continuous connectivity of the network is even more important and often mission critical.

Furthermore, applications usually run on small devices with very limited resources in terms of computation, communication and energy. In order to maximize the lifetime of the nodes, energy efficient algorithms are required on all layers of an application. One possibility for saving energy is to choose the right network topology. Especially in dense networks, where a node is within the transmission range of many other nodes, selecting an optimized subset of neighbors with which to communicate has a large potential to increase overall network performance and most important to reduce the power consumption.

A topology control algorithm that runs on wireless sensor nodes, selects a set of neighbors for communication, such that the resulting topology has advantageous properties. It can have several objectives. In this work the goal is to reduce the number of communication links and to avoid bad quality connections, while guaranteeing a robust network with redundant connections whenever possible. When G is the visibility graph, containing an edge for every *possible* link between two nodes, and G_{TC} the reduced graph obtained through topology control, with only the *selected* links, the connectivity is guaranteed if G_{TC} is connected, whenever G is connected (see Figure 29). Solving the topology control problem is challenging, because the algorithm needs to be fully distributed and only local information can be used for the decision-making in order to reduce traffic overhead. Furthermore a topology control algorithm should be simple and practical, a dominant prerequisite for successful implementation on commonly available sensor node platforms, i.e. additional hardware for localization should not be required.

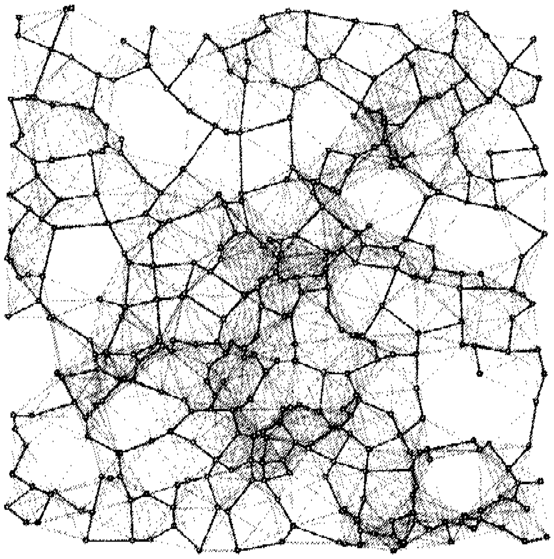


Fig. 29: Example with the graph G_{TC} with links selected by a topology control algorithm (bold edges) and the graph G with all possible links (light edges).

4.1.2 Bluetooth Networking

In our implementations, we use Bluetooth as the underlying communication infrastructure. Therefore, we give here a short introduction into Bluetooth networks. A more detailed discussion of Bluetooth networks and its usage for WSNs can be found in [Beu05].

Networking in Bluetooth is organized in master-slave configurations of up to seven active slaves that can be connected to one master at a time. This structure is called a *piconet*. Multiple piconets can be interconnected by nodes taking on dual roles of slave–slave or master–slave forming a *scatternet* (see Figure 30). While the interconnection of nodes in these different configurations is part of the Bluetooth standard, the formation and control of multi-hop topologies is not governed by the standard. Also, the data transport is only defined on each single hop (from master to slave or vice versa) and not over multiple hops, even within a piconet.

Bluetooth is a connection-oriented communication medium, i.e. in order to communicate with a neighbor, a node must explicitly establish a connection to it. For this procedure the node must first know its neighbors. Bluetooth devices use the asymmetric inquiry procedure to discover nearby devices by sending an inquiry request on the inquiry scan channel. Devices that are available to be found are known as discoverable devices and listen and respond to these inquiry requests (inquiry scan). Also the procedure for forming connections is asymmetrical and requires that one Bluetooth device carries out the connection procedure (paging) while the other Bluetooth device is connectable (page scanning). Neither the

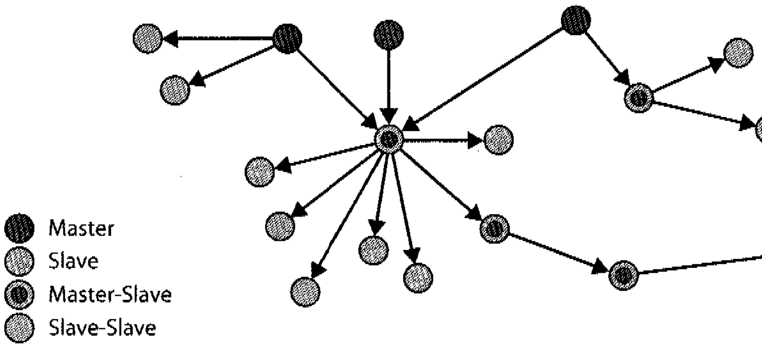


Fig. 30: Bluetooth scatternet with nodes in master-, slave-, master–slave-, and slave–slave roles.

inquiry nor the page operation give a guarantee for success. The user can specify a duration for these operation and during this time the Bluetooth device executes the procedures with best effort. However for the inquiry and page operation, durations in the order of a few seconds yield good results.

Once connected, data can be sent over a link using either asynchronous connectionless (ACL) or synchronous connection oriented (SCO) transfer. We use the ACL mode, which has an increased reliability because it includes forward error correction and retransmission of packets.

4.1.3 BTnut Connection Manager

In order to validate different topology control algorithms on the BTnodes, we designed a modular and abstract connection manager module that is integrated into the BTnut multi-hop system software [BBD+07]. The connection manager module acts on top of the *l2cap-connectionless*-layer, which is a simplified version of the *l2cap*-protocol that basically offers service multiplexing.

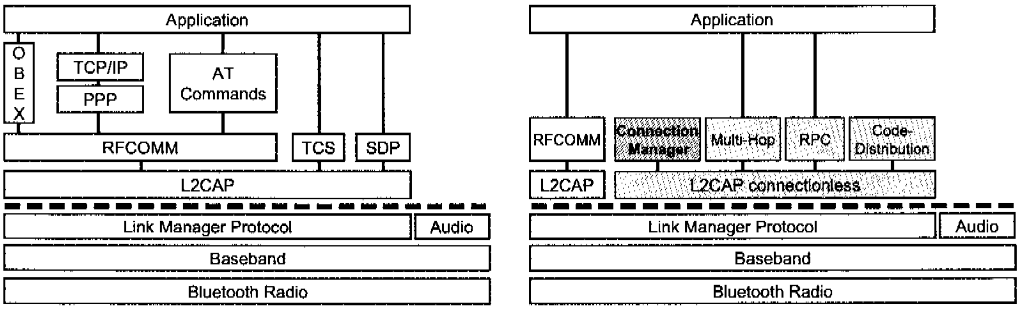


Fig. 31: Standard (left) and BTnut (right) Bluetooth Protocol Stack. The connection manager is an independent service on top of the l2cap-connectionless layer.

Figure 31 depicts for comparison the standard Bluetooth protocol stack on the left and the BTnut stack on the right. In the new l2cap-connectionless layer, services can be registered only once and are handled for incoming packets on all connections. In contrast, in l2cap (conneciton oriented), services are registered per connection, but only on a single link and not over multiple hops.

The connection manager module is responsible for setting up and maintaining a connected topology by discovering and connecting to other devices. When a new connection is established or an existing one is removed, the connection manager informs the higher layers and updates a shared neighborhood-table. Other modules, such as the multi-hop-routing module can then use this information for their service.

This simple abstraction allows us to write and test efficiently multiple topology control algorithms.

4.1.4 Topology Control — Related Work

Early work in topology control focused on the special case of randomly distributed nodes. Hou and Li [HL86] can be considered originators of topology control.

We do not consider centralized algorithms here, as they can not be implemented on distributed nodes, the typical scenario found in wireless sensor networks. In [RM99], distributed topology control algorithms are presented that assume that every node knows its location (e.g. from a GPS device). The CBTC algorithm [WLBW01] is based on directional information, which could also be achieved by directional antennas and beam-forming.

Topology control for Bluetooth ad-hoc networks is also referred to as scatternet formation. Previous research in this area has led to a number of algorithms, that can be distinguished according to the following three properties: (i) guarantee for a connected topology, (ii) guarantee for degree constraints, and (iii) the requirement that all devices are within each others range. In [BBMP04], the authors compared four scatternet formation algorithms and evaluated their performance. A more recent comparison with four additional algorithms can be found in [VGSR05]. According to the classification introduced earlier, only two of eight evaluated algorithms guarantee properties (i) and (ii) while not requiring all devices to be within each others range. One of them, the BlueMesh algorithm [PBC04], forms a mesh topology. However, it only considers connectivity information for neighbor selection. In contrast to BlueMesh, the second algorithm, described in [VGSR05] additionally uses RSSI as link-metric and constructs a sparse mesh scatternet, also known as the relative neighborhood graph (RNG).

The XTC algorithm [WZ04] uses an abstract link-metric, e.g. RSSI, to construct a graph corresponding to the RNG with the abstract link-metric as distance. Compared with previous solutions, XTC is probably the simplest, and therefore most practical topology control algorithm, that guarantees connectivity while not requiring all nodes to be within each others range. Additionally, XTC is a local algorithm, i.e. it does not require communication over multiple hops.

Algorithms such as the ones discussed above still have considerable drawbacks, that render them impractical for implementation. First, most algorithms assume that all nodes start execution of their protocols simultaneously. Second, especially the RSSI-based algorithms assume that the estimation of the distance using RSSI has a fidelity of 1, i.e. $\mathrm{RSSI}_1 > \mathrm{RSSI}_2 \Rightarrow d_1 < d_2$. In practice, this is not the case as will be shown later on.

In [BL03] the authors present an algorithm where a node connects to the k neighbors with the highest RSSI (k is a predefined constant). Although the connectivity cannot be guaranteed using this algorithm, it addresses the two issues mentioned above. The authors used a stochastic RSSI model in their simulations. Further, the execution of the algorithm is delayed in order to allow the nodes to start within a given time interval. However, this approach is still not flexible enough for practice, as nodes cannot join the network after the first links have been established.

4.2 Tree Algorithm

The first topology control algorithm constructs and maintains a tree structure. It is a robust and completely local algorithm that automatically takes care of nodes that join or leave the network. The basic principle is simple: Every node periodically searches for other nodes, and subsequently randomly connects to one of the discovered nodes. In parallel, detected cycles are removed.

The tree structure is constructed and maintained by a thread and two message handlers (see Alg. 3). The inquiry thread periodically performs an inquiry and randomly connects to one of the discovered devices. The message handlers process negotiation or tree-ID packets arriving from the lower layers. These packets are used to maintain the tree structure by preventing and detecting cycles in the network topology. All nodes connected in a tree share the same tree ID. When two nodes connect, they exchange negotiation messages and compare their tree IDs. If the two nodes were not in the same tree before the connection, their IDs differ and they have to establish a unique ID for the newly formed tree. This is done by agreeing on the larger of the two IDs and broadcasting it in

Algorithm 3 Tree algorithm

```
begin thread Inquiry
    loop
        $found\_nodes := inquiry(T_{inq})$
        $node := randomly\_select(found\_nodes)$
        $connect(node)$
        $send(node, negotiation(local\_tree\_ID))$
        $sleep(T_{sleep})$
    end loop
end thread
```

```
msg_handler $negotiation(remote\_tree\_ID)$
    if $local\_tree\_ID = remote\_tree\_ID$ then
        drop connection
    else
        if $local\_tree\_ID < remote\_tree\_ID$ then
            $local\_tree\_ID := remote\_tree\_ID$
            broadcast negotiation($local\_tree\_ID$) to my subtree
        end if
    end if
end msg_handler
```

```
on $disconnect$
    broadcast tree_ID($local\_tree\_ID$) to my subtree
```

```
msg_handler $tree\_ID(remote\_tree\_ID)$
    if $local\_tree\_ID = remote\_tree\_ID$ then
        drop connection
    else
        $local\_tree\_ID := remote\_tree\_ID$
        broadcast tree_ID($local\_tree\_ID$) to my subtree
    end if
end msg_handler
```

a tree-ID packet to all nodes in the subtree with the smaller ID. If two nodes that are already in the same tree connect, they will notice that they share the same ID and therefore drop the connection. If a node receives a tree-ID broadcast with an ID different from its own, it adopts this new ID. If the received ID is its own ID, there is a cycle in the network and the link over which the broadcast arrived is dropped. This mechanism eliminates cycles that can arise when two subtrees are connected almost simultaneously via two different links (see Fig. 32); in this case, the cycle prevention by negotiation does not work.

If a link is lost, the tree is partitioned, and the two subtrees must not share the same tree ID anymore. Therefore, if a node loses the link over which the current tree ID was received, the node broadcasts its unique device ID as its subtree's new tree ID.

This local algorithm provides self-healing topologies in a robust and completely distributed fashion. It does not need exhaustive computation

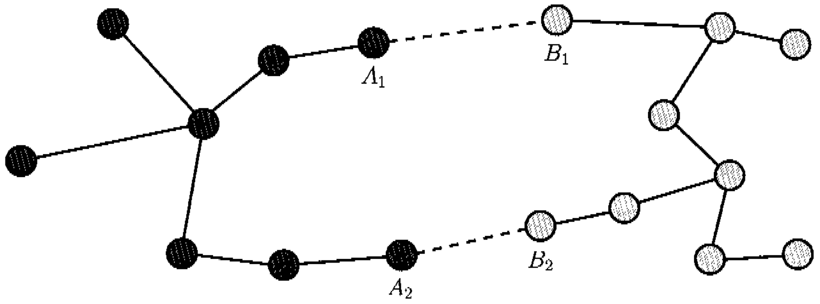


Fig. 32: Cycles can form when disconnected trees are connected almost simultaneously at two different points. The tree-ID broadcast eliminates the cycle shortly afterwards.

or communication. Therefore, it is expected to scale well to a large number of nodes.

4.2.1 Tree Algorithm Validation

We have tested and measured the properties of our implementation in two different setups. The first one was a lab setup involving 2–40 nodes on which we observed the network-topology construction. In the second setup, we distributed 71 BTnodes on a large office floor, thus obtaining a larger, realistic deployment scenario.

In both setups, all nodes are running the same software. A host PC is connected over a 115 kbps serial link to one of the BTnodes. This node receives topology information from the other BTnodes: Each node sends information about events such as new connections and link losses to the host, and additionally stores them in a local log. The logs are remotely collected by a monitoring and control application running on the host PC.

4.2.1.1 Network-Topology Construction

The topology construction depends on the ability to discover other nodes and to successfully connect to them. Since a-priori assumptions about the state of remote nodes cannot be made before an actual connection, these are highly non-deterministic operations. While a node is inquiring or connecting, it might not be discovered by others. Previous measurements have shown that inquiring is a time-consuming process requiring in the order of tens of seconds [KL01, WMF02] for a reliable discovery of all nodes. Experiments have shown that for the tree algorithm, short but frequent inquiries accelerate the formation of large network clusters. Our experiments were conducted with the following values: inquiries last 3.8 s and pauses between inquiries are chosen randomly (to avoid that all nodes inquire simultaneously) between 3 and 20 s.

The tree-construction algorithm is truly distributed. Since connections are established in parallel, the algorithm can be expected to scale well with an increasing number of nodes. We have verified this assumption with the following experiment.

Initially, n nodes are switched on one after the other. After all nodes are connected in a single tree, we simultaneously reset them with a broadcast command from the monitoring tool. This then provides a common time base for all nodes. All nodes log their connection and disconnection events, annotated with the time since the last reset. After all nodes are again connected, the monitoring tool retrieves these logs from all the nodes.

Figure 33 illustrates the evaluation of the tree-topology construction in a multi-hop network. Each plot represents the average of ten different experiments with the same number of nodes. After a boot-up phase of approximately 13 s, the first connections are established. At 20 s, close to 50 percent of all the connections are established, and at 70 s the construction is finished. These values are independent of the number of nodes involved.

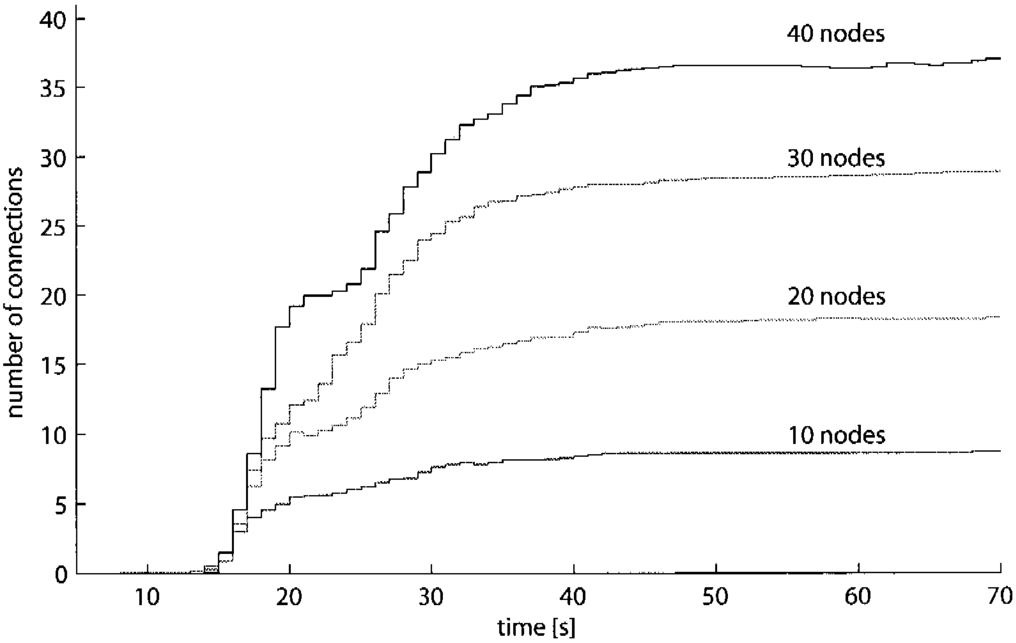


Fig. 33: Initial network-topology construction: Each curve represents the average of ten different experiments. The nodes were deployed with constant density.

All n nodes are connected in one tree if and only if there are $n-1$ connections. In some of the 40 experiments, not all n nodes were connected in the end. A software error in the low-level event-handling routines occasionally caused a deadlock in the inquiry thread (Alg. 3). As a consequence, some nodes were not able to discover and connect to other nodes anymore and remained isolated.

This was the reason why the above experiments were not conducted with more than 40 nodes. If many nodes are reset simultaneously, not all of them can connect in the first iterations of the inquiry thread, probably due to radio interference. Thus, the probability that a node's inquiry thread enters the deadlock before the node is connected increases with increasing node density. This problem is not inherent to the tree-algorithm and has disappeared with the low-level software errors resolved.

4.2.1.2 A Realistic Deployment Scenario

To test the tree-algorithm in a realistic scenario, we deployed 71 BTnodes on a large office floor. We also wanted to test our hypothesis that the problem described above should disappear if we reduce the node density.

We therefore distributed the nodes as depicted in Fig. 35, switching them on as we went along. Being switched on one after the other, all 71 nodes joined a single tree scatternet (see Fig. 34) without any problem. We then issued the reset command to all nodes. Within 70 s, 46 nodes had again connected to a tree. As more time passed, the tree did however not grow beyond this size. Essentially, the problem remained as severe as in the lab setup.

The explanation for this is that there is no sufficient difference in connectivity between the lab setup and the floor deployment. This can be seen in Fig. 35: the various connections over relatively long distances show that the average number of neighbors is still very high.

The quality of the long-distance links in Fig. 35 is smaller than that of the short-distance links. An improved algorithm would hence prefer high-quality links.

4.3 S-XTC: A Signal-Strength Based Topology Control Algorithm

The above discussed tree-based algorithm is extremly simple and efficient. It builds and maintains a connected sparse topology. However, its simplicity is also its limitation. With the random search and connect procedure, the algorithm does not distinguish between good and bad quality links. Furthermore, considering the connectivity constraints of Bluetooth devices, the algorithm can not always guarantee a connected topology. In this section, we investigate a more advanced algorithm, that addresses these issues.

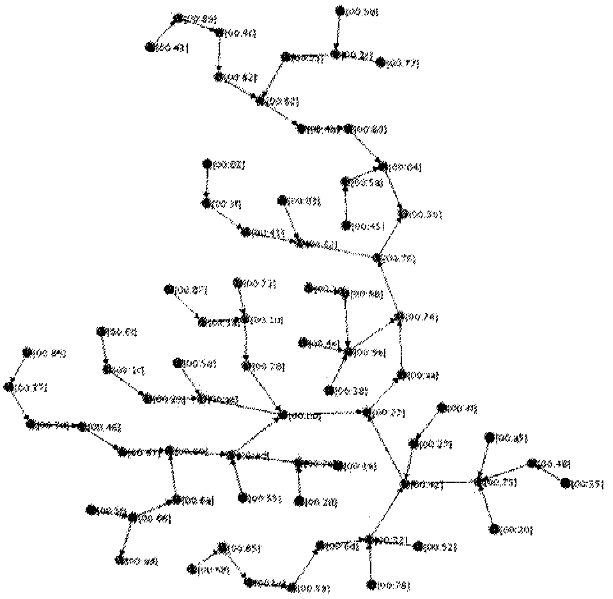


Fig. 34: A monitoring tool shows a scatternet tree topology with 71 nodes.

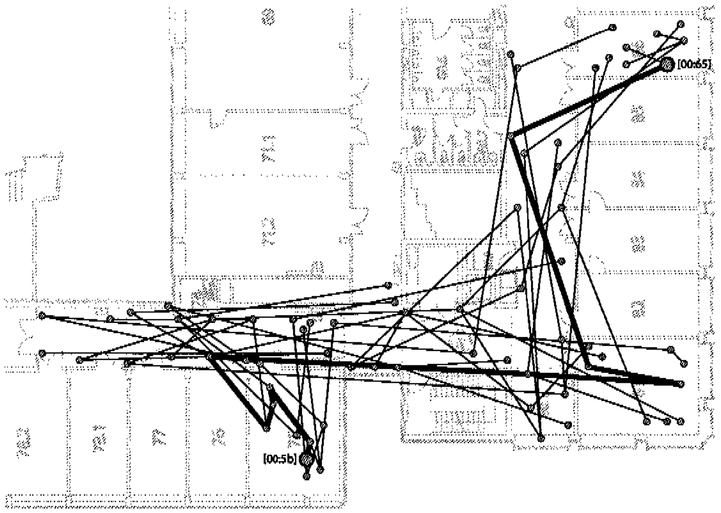


Fig. 35: The example shown in Fig. 34 was set up on a large office floor. The actual connections from Fig. 34 are shown.

4.3.1 Application Scenario

The applications scenario discussed assumes a distributed wireless sensor network application with a single node acting as a central data sink and multiple distributed data sources. The application requires a connected network topology and should exhibit some robustness against failures and changes in the connectivity of the nodes. This is quite a typical scenario commonly found in many wireless sensor network applications. Such a typical networking scenario for topology control is discussed in the following.

A large number of sensor nodes are deployed in a field. The nodes need to be deployed with sufficient density such that no disconnected clusters exist[1]. Besides that, no additional constraints on the deployment are made, which allows for both sparse and very dense regions.

In the scenario considered, the nodes remain stationary. However, it is possible that additional nodes join or occasionally leave accounting for cases of variable power supplies (e.g. batteries, solar-powered cell), failures or additional deployments at a later time. It is further possible that the quality of a connection between two nodes is temporally impaired by obstacles or other interference.

The nodes all have the capability to assess the strength of received signals at their radio. This is a reasonable assumption, as this feature has become a standard in almost all modern radio devices used on sensor nodes today.

When the nodes are turned on they start executing the topology algorithm which then runs forever or until a certain state is reached. Such a state can be characterized by a condition being met at a node, e.g. availability of a route to a base-station, or by more specific performance metrics. Upon detection of the loss of such a state, the topology algorithm can be re-enabled to further refine the network topology.

Topology control algorithms are typically implemented as a separate network layer in the embedded software of the nodes. The topology control layer notifies the higher layer about the links to the selected neighbors. The advantages for the application of the new topology control algorithm presented here are as follows:

- *Connectivity:* The application can reach all other nodes through the selected neighbors.
- *Low Degree:* Even in very dense networks the number of selected neighbors is low, which allows for more efficient execution of communication operations such as routing or flooding. Furthermore a

---

[1] As studied in percolation theory [DTH02], for randomly and uniformly distributed nodes, the network will be connected with very high probability if the network density is above 5 nodes per unit disk.

low degree is essential if protocols are used that limit the maximal number of connected neighbors.

- *Energy Efficiency:* A node can save energy in two ways through topology control: For link oriented communication systems, such as Bluetooth, the power consumption generally grows linearly with the number of connections [NBD06]. A lower degree then consequently leads to lower power consumption. For communication systems that can individually adjust the transmit power, energy is saved by reducing the transmit power to the level that is just needed to reach the worst selected neighbor and not the whole neighborhood anymore.

4.3.2 XTC Algorithm

Our algorithm is based on the recently published XTC algorithm [WZ04], which is an extremely simple and strictly local algorithm. The authors of XTC claim that it is faster than any previous proposal and that it is currently the most realistic topology control algorithm available. Let us briefly describe here the basic functionality of XTC.

The algorithm consists of three main steps: Neighbor ordering, neighbor order exchange, and edge selection (see Alg. 4).

Algorithm 4 XTC

```
Establish order $<_u$ over $u$'s neighbors in $G$
Broadcast $<_u$ to each neighbor in $G$; receive orders from all neighbors
Select topology control neighbors:
  $N_u := \{\}$; $\tilde{N}_u := \{\}$
  while ($<_u$ contains unprocessed neighbors) do
    $v :=$ least unprocessed neighbor in $<_u$
    if ($\exists w \in N_u \cup \tilde{N}_u : w <_v u$) then
      $\tilde{N}_u := \tilde{N}_u \cup \{v\}$
    else
      $N_u := N_u \cup \{v\}$
    end if
  end while
```

The algorithm operates on a initial graph $G = (V, E)$, which we refer here to as the *visibility graph*. For simulation and comparison, it is commonly accepted to assume that Graph G is a Unit Disk Graph (UDG), i.e. an Euclidean graph containing an edge (u, v) if and only if the normalized distance $|uv|$ is smaller or equal than 1. In other words: two nodes only "see" each other if they are within each others range.

In the first step, each network node computes a total order over all its neighbors with respect to decreasing link quality. This link ordered is then exchanged with all 1-hop neighbors. A node u selects the link to node v, if there exists no node w that has already been processed, that appears before u in the received order $\prec_v$ ($w \prec_v u$). After completion of the algorithm, the set N_u contains u's neighbors in the topology control graph G_{XTC}.

The main properties of the resulting topology of G_{XTC} are the following (see [WZ04] for additional properties and proofs):

1. *Symmetry:* A node u includes a neighbor v in N_u if and only if v includes u in N_v.

2. *Connectivity:* Two nodes u and v are connected in G_{XTC} if and only if they are connected in the visibility graph G. Consequently, the graph G_{XTC} is connected if and only if G is connected.

3. *Bounded Degree:* Given an Euclidean Graph G, G_{XTC} has degree at most 6.

4. *Sparseness:* In an average-case simulation where nodes are placed randomly and uniformly on a Euclidean plane, the average degree of the nodes in G_{XTC} is constant at approximately 2.5.

4.3.3 Signal-Strength Measurements

XTC has been evaluated in [WZ04] mainly on Euclidean graphs. When initially using RSSI values instead of the Euclidean distances as the link quality metric for the ordering of neighbors in the first step of the algorithm, we obtained differing results. In order to understand this behavior we first measured the behavior of the $RSSI_u(v)$ on our target platform.

The results are shown in Fig. 36 and Fig. 37. We see that the RSSI values, even in average, are not a monotonously decreasing function of the distance, which can most likely be attributed to multipath reflections and other sources of interference. A second effect is that in a stationary scenario with constant distance between a transceiver pair, the values measured over time have a large variance. Other radio transceivers are known to exhibit similar behavior [PSC05].

To investigate the consequence of these effects on the properties and the resulting topology we simulated the XTC algorithm. In contrast to the simulations presented in [WZ04] we use the following RSSI model for the ordering the neighboring nodes instead of the Euclidean distance:

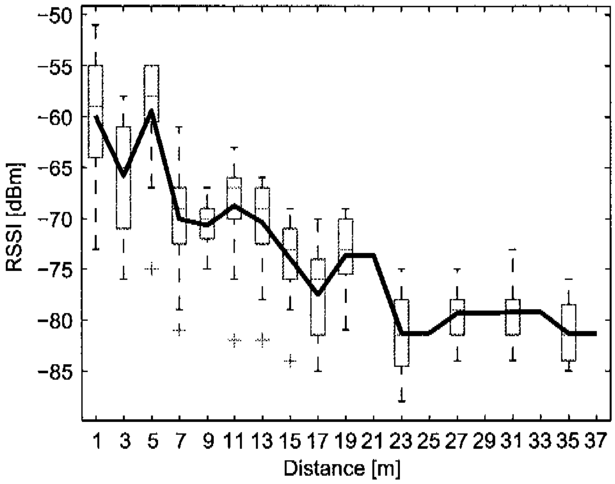


Fig. 36: RSSI over distance. The measurements are performed with two BTnodes at distance $|uv|$ and 1 m above the ground and alternated transmitting and receiving. The bold line is the average of 20 values. We read the RSSI of an unconnected neighbor with the hci_inquiry command (see HCI functional description of the Bluetooth Core 1.2. Specification). The boxes in the plot have lines at the lower quartile, median, and upper quartile values. The whiskers are lines extending from each end of the box to show the extent of the rest of the data. Outliers (+) are data with values beyond the ends of the whiskers.

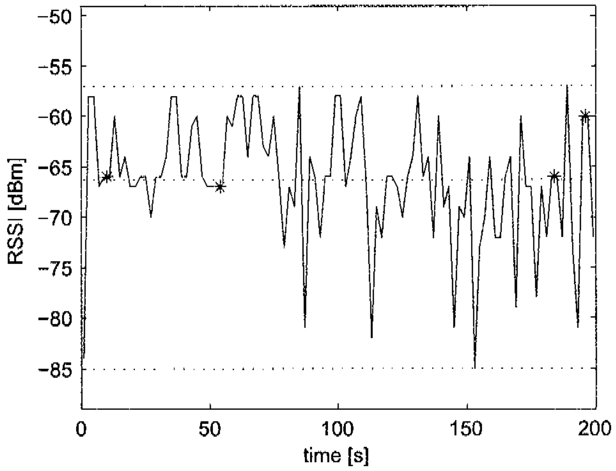


Fig. 37: RSSI fluctuations over time of two stationary nodes. The stars denote unsuccessful RSSI estimates (nodes were not found by hci_inquiry) that are commonly found in measurements. Most algorithms poorly account for such sporadic outages and thus are hard to transfer to realistic scenarios.

$$\begin{aligned} w <_u v &\Leftrightarrow RSSI_u(w) > RSSI_u(v) \\ RSSI_u(v) &= RSSI_1 - a \cdot \log(|uv|) - \sigma_{RSSI} \cdot x \\ a &= \frac{RSSI_1 - RSSI_{d_{\max}}}{\log(d_{\max})} \\ x &\sim \mathcal{N}(0,1) \end{aligned}$$

The reflections and the fluctuations of the RSSI are expressed in the model with the standard deviation σ_{RSSI} multiplied with a standard normal distributed random variable x. The model fits to our measurements with the values $RSSI_1 = -57\ dBm$, $RSSI_{d_{\max}} = -83\ dBm$, $d_{\max} = 40\ m$, and $\sigma_{RSSI} = 6\ dBm$. In the simulation 1000 nodes are placed randomly and uniformly in a square field. We altered σ_{RSSI} and the network density δ. The network density is defined as the average number of nodes within range ($d_{\max}$), which is equivalent to the number of nodes per unit disk. We simulated with densities from 1 to 30 nodes per unit disk and with a σ_{RSSI} from 0 to 13. The plots presented are averaged results of 1000 graphs.

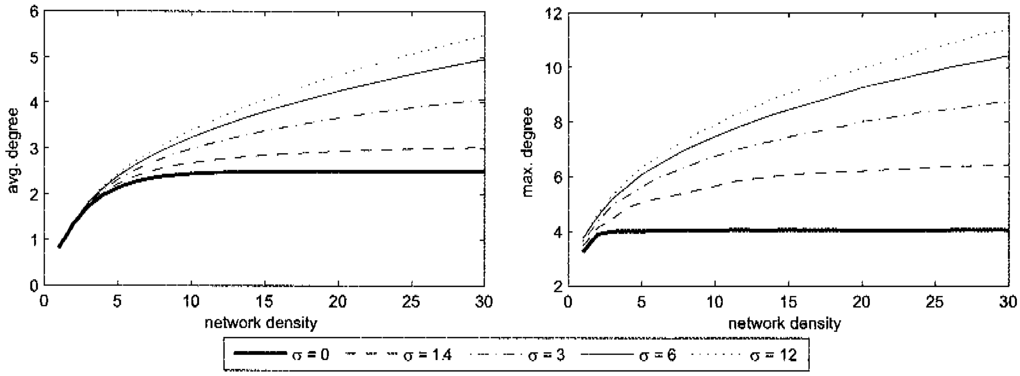


Fig. 38: Average and maximal degree vs. network density with varying RSSI standard deviation σ_{RSSI}.

Fig. 38 shows the average and maximal degree of the nodes in G_{XTC}. For $\sigma_{RSSI} = 0$ the neighbor ordering is the same as obtained with the Euclidean distance in [WZ04][2]. However for $\sigma_{RSSI} > 0$, the degrees are not bounded. They grow with network density and with σ_{RSSI}. The number of edges also increases, as it is proportional to the average degree. For larger σ_{RSSI}, the link metric becomes uncorrelated to the distance. It was shown in [WZ04] that for the case of arbitrary link weights (general weighted graphs) the degree of the nodes in G_{XTC} with n nodes can be $\Theta(n)$ and the number of edges $\Theta(n^2)$.

---

[2] $-a \cdot \log(|uv|) > -a \cdot \log(|uw|) \Leftrightarrow |uv| < |uw|$

4.3.4 S-XTC Algorithm

The XTC algorithm, as well as the majority of the other algorithms proposed, cannot be applied directly to our application scenario, because they contain rather impractical assumptions. In the case of XTC, we have identified the following problems.

1. Device Discovery is an unreliable operation. Especially in dense networks, it cannot be guaranteed that all neighbors are found, as required in the first step of XTC.
2. The nodes are not a priori synchronized and do not all start at the same time. However, an assumption in XTC is, that the three construction phases are synchronized on all nodes.
3. It is assumed that the topology does not change.
4. For the XTC algorithm, the link weights are required to be symmetric. However, without additional communication we have $\mathrm{RSSI}_u(v) \neq \mathrm{RSSI}_v(u)$, and no guarantee for the connectivity can be given anymore.
5. A realistic RSSI is not a monotonous decreasing function of the distance ($\mathrm{RSSI}_u(v) > \mathrm{RSSI}_u(w) \Leftrightarrow |uv| < |uw|$). The maximal degree of a node is not bounded in that case.
6. As in XTC, every node needs to exchange its ordered list with all reachable neighbors, the time needed for this operation scales badly with the network density.

Our proposed S-XTC algorithm addresses the above problems by providing 3 extensions.

- *Dynamic Adaptation:* With a general protocol change we solve problems 1-4.
- *Bounded Degree:* This extension can be used if a bounded degree is required (Problem 5).
- *Scalability:* With an additional heuristic, we reduce the number of required ordered list exchanges, while still guaranteeing the connectivity (Problem 6).

4.3.5 Wireless Communication

4.3.5.1 Communication Primitives

In order to foster an implementation of our algorithm not only for simulation but also on real sensor nodes, we give a more detailed description of the algorithm than such presented in related work. In particular, we describe what messages are exchanged and how they are handled. We use the following communication primitives:

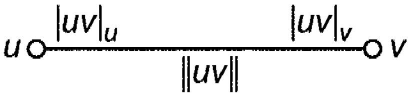
discover: Upon request, beacon messages with the device ID are sent at the remote node. When such a message is received, the remote ID is stored and the RSSI value is measured. For Bluetooth, an integrated function (hci_inquiry) exists to perform this function.

connect/disconnect: This is only needed in a connection oriented medium such as Bluetooth, where a dedicated channel between two nodes must be established before messages can be sent.

send: We only consider point-to-point communication. The reason is that in dense networks the high interference would make a reliable broadcast transmission hard to achieve. In the case of Bluetooth, the communication is connection oriented.

4.3.5.2 Asymmetric RSSI link model

The algorithm uses a link-metric to decide whether a link is selected or not. However, if only local RSSI values are incorporated into this decision the resulting network topology might not be fully connected. The reason for these asymmetries is that the RSSI values measured by an adjacent node pair are usually not identical on both sides of the link. Therefore, we need to define a common link-metric. We use the following notation for a link in the visibility graph G



$$\begin{aligned} |uv|_u &= -\mathrm{RSSI}_u(v) \\ |uv|_v &= -\mathrm{RSSI}_v(u) \\ \|uv\| &= f(|uv|_u, |uv|_v) \end{aligned}$$

where $\mathrm{RSSI}_u(v)$ is the RSSI value that node v measures to node u. Increasing RSSI values denote increasing link quality. $\|uv\|$ is the combined link-metric derived from the exchange of the RSSI values at each node pair. It can be calculated using the function f on each side of a link as soon as the other value has been received. An evaluation using different functions for f will follow in section 4.3.8.

4.3.6 Dynamic Adaptation

The pseudo-code for this extension is given in Algorithm 5.

The main idea is that the decision, whether node u selects neighbor v or not, is taken directly when an ordered list $\prec_v$ is received at u. This results in the same selection as in the original XTC algorithm, because the decision only depends on $\prec_u$ and $\prec_v$ and not on other neighbors' ordered lists. Furthermore, this allows for asynchronous operation at the nodes. Decisions can be taken independently from other neighbors' state and directly on arrival of a neighbor's ordered list. A further advantage is that the algorithm increases memory efficiency. In this way, there is no need to store ordered lists from all neighbors. In fact received lists can be discarded right after a single selection decision has been taken.

The algorithm has three processes that run in parallel. The *discovery* process is responsible for detecting network changes such as newly discovered nodes or changes in the link quality. The RSSI from all visible neighbors are repeatedly measured and stored with the corresponding node ID in a ordered list $\prec_u$. Based on the old and the new list, it can be determined which neighbors are affected by the change. Those neighbors are added to an ordered queue Q. It can not be guaranteed that every time all neighbors are correctly detected. The algorithm therefore continuously repeats these operations which ensures statistically, that all neighbors are found if the algorithm runs long enough.

The *connect* process continuously processes the entries in Q by sending $\prec_u$ to the neighbors. In contrast to the original XTC algorithm, the list $\prec_u$ not only contains the order of the nodes, but also the absolute link weight, which are needed later to calculate a combined symmetric link-metric.

The *message dispatch* process calls the appropriate handler for every incoming message. If an ordered list is received the function *makeSymmetric* is called. The combined link-metric $||uv||$ can be calculated from the local link-metric in $\prec_u$ and $\prec_v$. The nodes v and u are then reinserted according to $||uv||$ into $\prec_u$ and $\prec_v$ respectively. This reordering however, can influence the selection decision of the nodes w (line 5 in *makeSymmetric*) that are between v's old and new position in $\prec_u$. Finally, these nodes are then reinserted into the queue Q for processing.

4.3.6.1 RSSI Fluctuations

The algorithm reacts on network changes such as node addition and deletion and link losses. However, unnecessary changes due to RSSI fluctuations should be avoided as much as possible, since they cost energy and impair network stability. In a field experiment with stationary nodes, deployed with 12 nodes per unit disc, we counted the number of connection establishments. Filtering the RSSI with a low-pass filter

Algorithm 5 Adaptive XTC (for an individual node *u*)

```
process discovery
    repeat
        discover network changes
        determine order $\prec_u$
        add changed neighbors to ordered queue $Q$
        sleep($t_{\text{sleep}}$)
    until termination event
end process
```

```
process connect
    loop
        if ($Q$ not empty) then
            $v$ = next node in $Q$
            connect(v) // if needed
            send($v$, orderlist($u$, $\prec_u$))
            remove $v$ from $Q$
        end if
    end loop
end process
```

```
msg_handler orderlist(node $v$, list $\prec_v$)
    remove $v$ from $Q$
    makeSymmetric($\prec_u$, $\prec_v$)
    if ($\exists w : w \prec_u v$ and $w \prec_v u$) then
        send($v$, nack($u$, $\prec_u$))
    else
        send($v$, ack($u$, $\prec_u$))
        inform higher layer that $v$ is selected
    end if
end msg_handler
```

```
msg_handler ack(node $v$, list $\prec_v$)
    makeSymmetric($\prec_u$, $\prec_v$)
    inform higher layer that $v$ is selected
end msg_handler
```

```
msg_handler nack(node $v$, list $\prec_v$)
    makeSymmetric($\prec_u$, $\prec_v$)
    inform higher layer that $v$ is deselected
    disconnect($v$)
end msg_handler
```

```
function makeSymmetric(list $\prec_u$, list $\prec_v$)
    determine $||uv||$
    $\prec_{u,\text{old}} = \prec_u$
    update $\prec_u$ and $\prec_v$ using $||uv||$
    add all ($w : w \prec_{u,\text{old}} v$ and $v \prec_u w$) to $Q$
end function
```

reduced the number of connection establishments by 20%. Additionally, we introduced a threshold value $\Delta RSSI_{th}$. The local neighbor order is only changed if the difference between the old and the new RSSI value is bigger than $\Delta RSSI_{th}$. Fig. 4.3.6.1 shows that an increased $\Delta RSSI_{th}$ stabilizes the network.

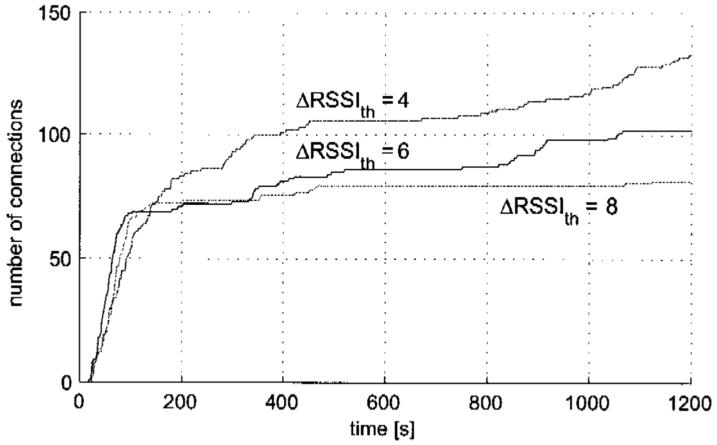


Fig. 39: Network establishment with 3 different $\Delta RSSI_{th}$ values in [dBm].

4.3.6.2 Evaluation on Testbed

We have implemented and evaluated S-XTC on a BTnode testbed with up to 40 nodes. Each node in the testbed locally stores the progress of the algorithm in a logfile, which is then retrieved by the base-station for evaluation. In order to measure the connectivity, we have added a process that periodically broadcasts a ping packet that is flooded on the network.

Fig. 40 shows an example logging history of the S-XTC start-up on a node that has 14 neighbors. We have measured the time needed until the whole network is connected (dashed line), the time until all nodes are discovered (bold line), and the time needed to exchange the neighbor orders with all visible neighbors (solid line). In different test runs, we have deployed the nodes with different node densities. Average values, that have been found are listed in Table 8.

Considering that with Bluetooth, operations such as *inquiry* and *connect* itself require typically a few seconds to complete, and that the nodes are not synchronized, the S-XTC algorithm achieves quickly a connected topology. However, the device discovery and interferences are slowing down the exchange of neighbor orders, especially in dense networks.

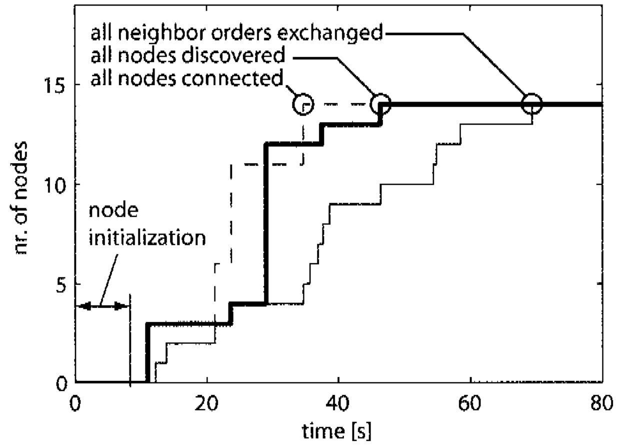


Fig. 40: S-XTC start-up measured on a single node running in a testbed with 15 nodes within transmission range.

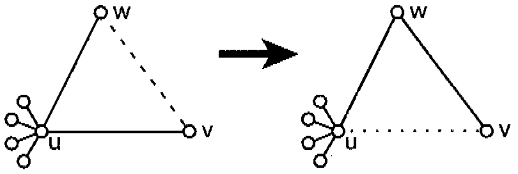


Fig. 41: When node *u* receives the ordered list $<_v$ it remembers node *w* as an alternative. If the degree of *u* is higher than the bound, the worst connection with an alternative is deleted. In this case *u* sends a message *swap(u, w)* to *v* to achieve the goal of global connectivity.

4.3.7 Bounded Degree

The problem of the unbounded degree is not solved by the first extension proposed. The discovery process measures the signal strength to neighboring nodes periodically and averages the result. This has the effect that fluctuations are smoothed and the correlation between RSSI and distance is improved. Experiments have shown that we could reduce the standard deviation σ_{RSSI} by filtering from 6 to 3 dBm. However, it can be seen from Fig. 38, that for $\sigma_{RSSI} = 3$ the maximal degree being found using simulation is still above 8 for high densities. If nodes have limited number of possible connections such as the case on Bluetooth [KAH+04], a guarantee for the connectivity of the network topology derived cannot be given. We therefore propose a further extension using Alg. 6.

This algorithm replaces the message handler *orderlist* and *ack* of Algorithm 5 and adds a further handler *swap*.

We will now explain this algorithm using the example shown in Fig. 41. An ordered list $<_v$ is sent to node *u*, which starts to iterate through this list. Node *u* searches for common neighbors *w*, that appear before *v* in $<_u$. If

Algorithm 6 Bounded XTC (for an individual node u)

```
1: process connect
2:     loop
3:         if ($Q$ not empty) then
4:             $v$ = next node in $Q$
5:             connect(v) // if needed
6:             send($v$, orderlist($u$, $\prec_u$ \ deleted($v$))
7:             remove $v$ from $Q$
8:         end if
9:     end loop
10: end process
```

```
1: msg_handler orderlist(node $v$, list $\prec_v$)
2:     remove $v$ from $Q$
3:     makeSymmetric($\prec_u$, $\prec_v$)
4:     repeat
5:         $w$ = next node in list $\prec_v$
6:     until (($w \prec_u v$) or endOfList)
7:     if (endOfList) then // no common neighbor in $\prec_v$
8:         degree = degree + 1
9:         send($v$, ack($u$, $\prec_u$))
10:        inform higher layer that $v$ is selected
11:    else if ($w \prec_v u$) then // better common neighbor
12:        send($v$, nack($u$, $\prec_u$))
13:    else
14:        degree = degree + 1
15:        send($v$, ack($u$, $\prec_u$))
16:        inform higher layer that $v$ is selected
17:        altNode($v$) = $w$
18:        if (worstCandidate $\prec_u$ $v$) then
19:            worstCandidate = v
20:        end if
21:    end if
22:    if ((degree > bound) and (worstCandidate not null)) then
23:        send(worstCandidate, swap($u$, altNode(worstCandidate)))
24:        deleted(altNode(worstCandidate)) = worstCandidate
25:        degree = degree - 1
26:    end if
27: end msg_handler
```

```
1: msg_handler ack(node $v$, list $\prec_v$)
2:     makeSymmetric($\prec_u$, $\prec_v$)
3:     degree = degree + 1
4:     inform higher layer that $v$ is selected
5:     repeat
6:         $w$ = next node in list $\prec_v$
7:     until (($w \prec_u v$) or endOfList)
8:     if ($w \prec_u v$) then
9:     ... same as orderlist line 17-25
10: end msg_handler
```

```
1: msg_handler swap(node $v$, node $w$)
2:     deleted(w) = v
3:     add $w$ to $Q$
4:     inform higher layer that $v$ is deselected
5:     degree = degree - 1;
6:     disconnect($v$)
7: end msg_handler
```

node initialization time:
~ 9 s
time until all nodes are connected:
~ 33 s (independant of density)
time until all neighbor orders are exchanged:
~ 60 s (density 10)
~ 122 s (density 15)
~ 256 s (density 20)
~ 414 s (density 25)
time to add a new node to a existing network:
~ 11 s
time for a newly added node to exchange all neighbor orders:
~ 36 s (density 10)
~ 62 s (density 15)
~ 90 s (density 20)
~ 198 s (density 25)

Tab. 8: Typical delay measurements of S-XTC.

no such node is found u selects v. Otherwise it is checked whether node w appears before u in $<_v$. If true, w is a better neighbor to both u and v, and the link uv is not selected. These two cases also appeared in the previous two algorithms. However the third case is new: if w appears after u in $<_v$, we have the situation shown on the left of Fig. 41. Node u can also reach v over w. This information is stored in an additional array *altNode*. Additionally, node u remembers the connected neighbor ranked least that has an alternative route *worstCandidate*. At the end of the message handler the current node degree is checked. If it is greater than the bound specified, it sends a *swap* message to *worstCandidate*. Node v handles the *swap* message, by adding node w to the queue Q and disconnecting the link to u.

The nodes of the disconnected link are not deleted from from the ordered list, because if found again during the discovery process, it would interpret them as new nodes. Instead, the nodes are only removed when sending an ordered list to an alternative node. Considering the example shown in Fig. 41 on the right side. When node u sends an ordered list to w (connect process, line 6), it sends $<_u \setminus v$; for all other nodes it sends $<_u$. So the edge uv is only deleted in the view of w. This information is stored in the array *deleted*.

Note that in the *swap* handler, the neighboring node w is not directly connected. Instead, it is added to the queue Q, in order to force node w to check if the new edge is really necessary.

The following theorem proves that this algorithm guarantees a maximal degree of 6, if G is a Unit Disk Graph.

Thm. 1: **(Bounded Degree)** *Given a Unit Disk Graph, the topology control graph,*

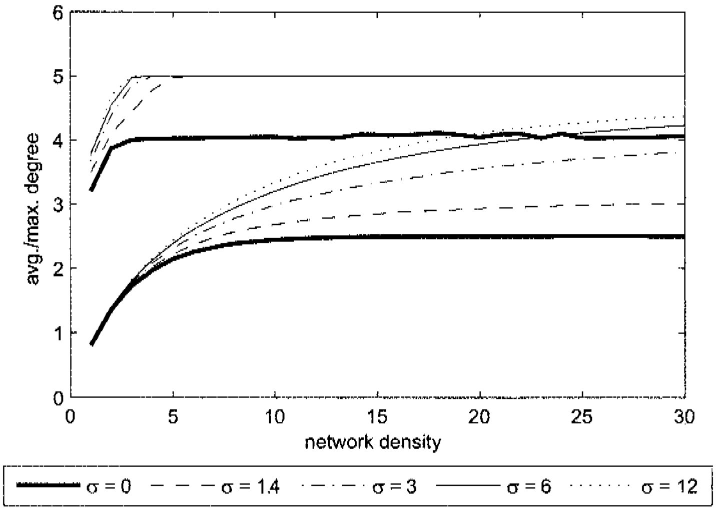


Fig. 42: Average (lower set of curves) and maximal (upper set of curves) degree of the Alg. 6 with a specified bound of 5.

obtained by Alg. 6 and a specified bound of 6, has at most degree 6.

Proof. Assume for contradiction that one node has degree bigger than 6. When the seventh neighbor was selected *worstCandidate* must have been *null* at the end of the *orderlist* or the *ack* message handler. This can only be the case if during the connection of all 7 neighbors no alternative node was found. Then this is only possible if there is no pair in the 7 neighbors that are also neighbors of each other. In a Unit Disk Graph, this would mean that no two adjacent edges enclose an angle less than $\pi/3$, which is only possible with at most 6 neighbors.

□

We have evaluated this extension with a specified bound of 5 in the same simulation setup as given for Fig. 38. The result is shown in Fig. 42, where we can see that none of the nodes violate the given bound. Compared to Fig. 38 the average degree is also improved, which is a result of the edges deleted.

In the simulation, where the Unit Disk Graph model is used, the degree bound can be set to 5. Note that in a real environment with interference, reflections, and obstacles, the UDG model is not realistic. In a worst case scenario, where a node has N neighbors, which do not see each other because of obstacles, the degree can be as large as N. However, this worst case too is an unrealistic assumption that could not be observed in test scenarios.

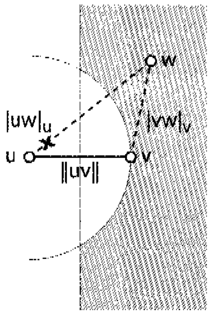


Fig. 43: Reduction of possible links for saving expensive exchanges: node *u* does not connect to node *w* if node *w* is in the shaded area.

4.3.8 Improved Scalability in Dense Networks

The third extension addresses the scalability in dense networks. Suppose for example that a node has 100 neighbors. The XTC algorithm would require to exchange the ordered list with all of them, even if at the end maximally 5 nodes are selected. There will be a high interference, because all the 100 neighbors also want to exchange their order with each other. Here, broadcast communication is clearly inferior. However for link-oriented communication, such as in Bluetooth, the situation is even worse. For every neighbor a connection has to be opened prior to the exchange and closed thereafter if it is not needed to support the network topology. The time for a connection setup in Bluetooth is in the order of a few seconds, if both nodes are ready. But if one node is already trying to connect to another node or performing a device discovery (inquiry), the delay can grow to tens of seconds, which is quite impractical. We have measured the time for the initial setup and found it to grow quadratically with the network density. This problem was also identified by the authors of [VGSR05].

4.3.8.1 Reduction Heuristic

With an additional heuristic, we can considerably reduce the number of exchanges required as shown in Fig. 43. Here, we use the notation of Sec. 4.3.5.2.

Suppose every node has a candidate list with all neighbor nodes, with which order lists have to be exchanged. This candidate list contains initially all neighbors. However, as soon as the first order lists have been exchanged, the candidate list can be reduced based on the received information. E.g. after node *u* has received the order list $\prec_v$ from *v* and calculated the common link weight $\|uv\|$, it can use additional information about a common neighbor *w* for the decision whether *w* is removed from the candidate list. In particular, node *u* removes node *w* from the candidate list, if

$$|uw|_u > |vw|_v \quad \text{, and} \tag{4.1}$$

$$|uw|_u > \|uv\|. \tag{4.2}$$

For Euclidean graphs, this is the case if node w is in the gray area in Fig. 43. An exchange of order lists between two nodes u and w is only prevented, if both u removes w and w removes u from their candidate lists.

We now show in the following theorem that with the proposed reduction applied to the XTC algorithm the connectivity is still guaranteed, if the common link weight is defined as $\|uv\| = \min(|uv|_u, |uv|_v)$.

Thm. 2: **(Valid Reduction)** *The reduction of the ordered list exchanges applied to the XTC algorithm does not lead to a disconnected topology if the common link weight is defined as* $\|uv\| = \min(|uv|_u, |uv|_v)$.

Proof. Since we know that G_{XTC} is connected, we prove that $G_{\text{XTC}} \subseteq G_{\text{reduction}}$. If an edge $uw \in G_{\text{XTC}}$ is not in $G_{\text{reduction}}$, it has not been selected due to a missing ordered list exchange. The only two possibilities for this are: (a) the edge uw is removed from the candidate lists of both u and w, and (b) the edge is not selected because another ordered list is missing.

Consider for contradiction an edge $uw \in G_{\text{XTC}}$ with

$$|uw|_u \leq |uw|_w \Rightarrow \|uw\| = |uw|_u. \tag{4.3}$$

In order to remove w from u's candidate list, there must have been an ordered list exchange between u and a third node v. Because $uw \in G_{\text{XTC}}$, the following equation must hold:

$$(\|uw\| < \|uv\|) \cup (\|uw\| < \|vw\|) \tag{4.4}$$

The conditions for the removal of w from u's candidate list are given in Eq. 4.1 and Eq. 4.2. Combining Eq. 4.1-4.3 we get:

$$\|uw\| > |vw|_v \tag{4.5}$$

$$\|uw\| > \|uv\| \tag{4.6}$$

With Eq. 4.4 and Eq. 4.6, we obtain:

$$\|uw\| < \|vw\| \tag{4.7}$$

Eq. 4.5 and 4.7 however, contradict the minimum definition:

$$\|uw\| < \|vw\| \leq |vw|_v < \|uw\| \tag{4.8}$$

This proves that (a) is not possible. For (b), consider the ordered list $<_u$ of a node u. A neighbor v can only be deselected due to a missing ordered list of a third node w, if w appears before v in $<_u$, but after v if $\|uw\|$ was known. This would mean that $|uw|_u < \|uw\|$ which is in contradiction to the definition.

□

4.3.8.2 Evaluation on Testbed

We have measured the startup times of S-XTC with the reduction heuristic and compared them to the previous results (see Fig. 44). We compare the time needed to exchange the neighbor order with all visible neighbors when all nodes have started synchronously (first two bars) and when only one node is added to an already existing network (last two bars).

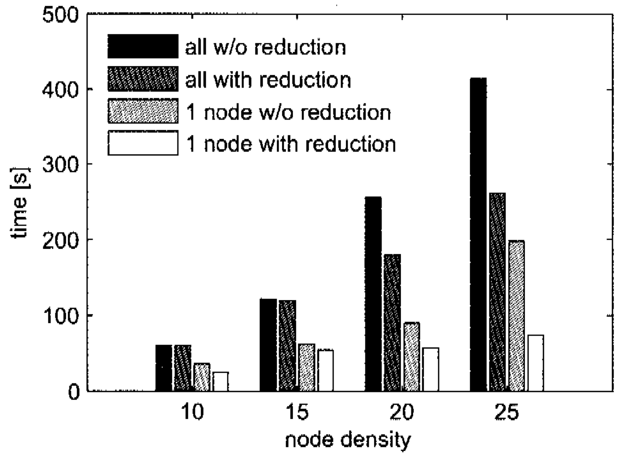


Fig. 44: Comparison of S-XTC startup time with and without the reduction heuristic.

For low densities, the startup time is dominated by the success of the device discovery and therefore the reduction heuristic has not much effect. With increasing node density the amount of necessary list exchanges among neighbors is reduced. We have observed that the number of effective exchanges with a node density of 25 is between 3 and 11, i.e. the reduction is on the order of 50%.

4.4 Case Study

The deployment-support network (DSN), which is descibed in the next chapter requires topology control. We use this applicatione here as a case-study for validating the S-XTC algorithm. The DSN is a wireless cable replacement for the development, testing and debugging of sensor network applications. As a general service, the DSN provides access to the sensor nodes with a wireless backbone network. One or more host computers are used to connect to the network and to communicate with the target sensor nodes using an underlying connected network topology.

In the first version of the DSN, the topology was constructed and maintained using the tree algorithm. The main problem with this solution was, that a link failure in the tree results in a disconnected network.

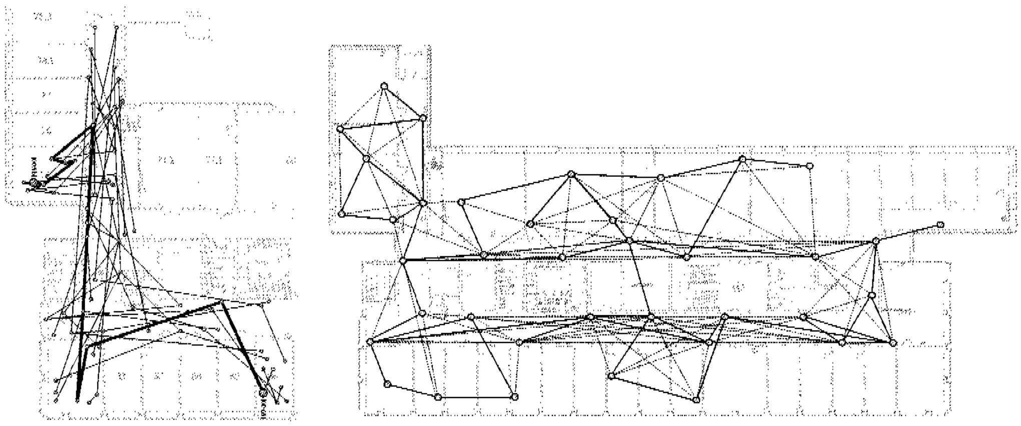


Fig. 45: Deployment of the DSN on an office floor using a previous tree based algorithm (left figure) and the new S-XTC algorithm using an RSSI link-metric (right figure).

Furthermore, because the tree algorithm is based on random search and connect, a number of low quality links are used in the backbone network resulting in sub-optimal topologies and significantly degraded reliability. With link failures happening at random, and in the worst case disconnecting the whole DSN from the host, the lack of redundancy can cause substantial disruption of service to the application.

We have successfully integrated S-XTC topology control in an implementation of a Deployment-Support Network using the BTnode platform. The application has been deployed on a testbed in an office floor comprising up to 50 nodes (see Fig. 45) and in various test scenarios (see Section 4.3.4 for details). Scenarios with differing numbers and densities of nodes have been successfully tested. The network density in the setup shown in Fig. 45 varies between 2 and 12. The algorithm successfully connected the nodes into a connected network with an average degree of 2.9 and a maximal degree of 5.

The capabilities of the S-XTC network to adapt to gradual changes in the network and the environment as well as its improved resilience to link failures has improved the overall stability of the DSN application.

4.5 Summary and Discussion

In this chapter we have analyzed existing approaches for network topology control and identified the key properties and eminent problems. Based on this analysis we have presented a practical topology control algorithm that combines:

- the guarantee for connectivity,

- energy efficiency through low degree topology control based on a single link metric,
- the ability to dynamically adapt to network and environmental changes, and

The S-XTC algorithm is based on three extensions to the original XTC algoritm, that render it into a reliable and practical topology control algorithm. The first extension *dynamic adaptation* addresses most shortcomings of the previous algorithm, i.e. operation in an unreliable environment, asynchronous startup and asymmetric link weights. The second extension *bounded degree* can guarantee an analytically proven bound on the node degree. The third extension *scalability* finally, significantly reduces the message overhead and computational complexity in dense networks.

The different extensions have been validated by means of analytic proofs and simulation. The simulation scenarios used have on the one hand been chosen similarly to the ones in the original XTC paper as to allow direct comparison and on the other hand to determine parameterizations and characteristics of the algorithm to facilitate implementation.

With the preparatory work from analysis, algorithm development and simulation at hand, and a preliminary test implementation of XTC, the S-XTC algorithm has been successfully implemented, on tiny, resource constrained wireless sensor network nodes.

The DSN application running on top S-XTC has proven the practicality of the algorithm in the case study presented in section 4.4. Benchmarks and field tests have demonstrated its performance: an increase in efficiency, lower message overhead, an improved selection of links as well as improved scalability over previous solutions. Even in high density scenarios (over 25 nodes within visibility) the time a first completion of S-XTC is reduced by approximately 1/3 over the algorithm without the proposed extensions. Likewise the amount of necessary neighbor list exchanges is reduced by about 50%.

Practical applications benefit from S-XTC due to advantages in the connectivity, low node degree and overall energy efficiency of the topology control algorithm.

Starting with a simple tree algorithms has been a very encouraging experience due to its traceable and comprehensible nature. The increase in complexity when actually implementing an algorithm is not to be underestimated. In conjunction with the complex behavior of the devices, the exact interpretation of an effect can be very hard.

In current research, there is an apparent gap between the results of theoretical and practical work [KMW04]. Seemingly simple algorithms often rely on the availability of complex functions which are not readily supported by the actual hardware. For instance, a function such as *send to all neighbors*, which frequently appears in algorithm descriptions, typically involves multiple operations such as *search for all neighbors, open a connection, send the data,* and *close the connection.* In practice, there is no guarantee for the success of these operations. Thus, a practical algorithm accounts also for imperfections and failures.

We have shown in this chapter that we require the high quality feedback from an implementation based validation in order to learn about and account for the complex physical phenomena that appear in a real-world scenario. This supports the thesis that for distributed embedded systems, adapted validation strategies are required that allow for testing in a realistic environment.

5

Deployment Support Network

In the previous chapters, we applied adapted methods for the validation of distributed embedded systems. We have shown that, especially for wireless sensor networks, the validation is very challenging, because of the limited resources of the nodes and the uncertain stimuli. Considering further the distributed nature of the algorithms, the large number of nodes, and the interaction of the nodes with the environment, we see that the nodes are also hard to access.

Access to the state of the nodes, referred to as visibility, is fundamental for validation. More visibility means faster development. But not only the amount of state information, but also the quality is important. Simulators for sensor networks for example, provide almost unlimited visibility. But on the other hand they use simplistic models for the abstraction of the environment. They fail to capture the complex physical phenomena that appear in real deployments. Therefore, the visibility of simulations is of lower quality.

For this reason, researchers have built emulation testbeds with real devices. Existing testbeds consist of a collection of sensor nodes that are connected to a fixed infrastructure, such as serial cables or ethernet boxes. Testbeds are more realistic than simulators because they use the real devices and communication channels. The problem that remains is that the conditions in the field where the WSN should be deployed in the end can be significantly different from the testbed in a laboratory. In particular, with a cable-based infrastructure it is almost impossible to test the application with a large number of nodes out in the field.

In evaluations of real deployment experiments like the ones presented in [SPMC04, TPS$^+$05, DHJ$^+$06], a gap between simulation-/emulation-results and the measured results of the real deployment has been reported. Measured packet yields of 50%, reduced transmission ranges, dirty sensors and short life-times of nodes did not match the expectations and the results obtained through simulation and emulation. The unforeseen nuances of a deployment in a physical environment forced the developers to redesign and test their hardware- and software components in several iterations. It has also been shown that sacrificing visibility, such as switching off debugging LEDs on the nodes in favor of energy-efficiency is problematic during the first deployment experiments [LBV06].

We introduce the *Deployment Support Network*, a toolkit for developing, testing and monitoring sensor-network applications in a realistic environment. The presented methodology is a new approach, since it is wireless and separates the debugging and testing services from the WSN application. Thus it is not dependent on a single architecture or operating system. In contrast to existing approaches, our method *combines* the visibility of emulation testbeds with the high quality of information that can only be achieved in real deployments. The DSN has been implemented and applied in an industrial case-study.

The chapter is organized as follows: section 5.1 presents related work, sections 5.2 and 5.3 describe our approach and its realization. In section 5.4 an industrial case-study is presented and finally, in section 5.5, we discuss the advantages and limitations of our method.

5.1 Development Support and Testing of WSNs – Related Work

To support the development and test of sensor-network applications various approaches have been proposed. On the one hand, simulation and emulation testbeds allow for observation of behaviour and performance. On the other hand, services for reprogramming and remote control facilitate the work with real-world deployments.

Simulation

Network simulators such as *ns-2* [ns2] and Glomosim [ZBG98] are tools for simulation of TCP, routing, and multicast protocols over wired and wireless networks. They provide a set of protocols and models for different layers including mobility, radio propagation and routing. TOSSIM [LLWC03] is a discrete event simulator that simulates a TinyOS mote on bit-level. TOSSIM compiles directly from TinyOS code allowing experimentation with low-level protocols in addition to top-level application systems.

Emulation Testbeds and Hybrid Techniques

Indoor testbeds use the real sensor node hardware which provides much greater sensing-, computation-, and communication realism than simulations. In some testbeds the nodes are arranged in a fixed grid, e.g. on a large table or in the ceiling. They are connected via a serial cable to a central control PC. In the MoteLab testbed [WASW05], each node is attached to a small embedded PC-box that acts as a serial-forwarder. The control PC can then access the nodes via the serial-forwarders via ethernet or 802.11.

The *EmStar* framework [EGE04] with its *Ceiling–Array* is also an indoor testbed. It additionally provides the ability to shift the border between simulation and emulation. For instance, the application can run on the simulator whereas for the communication the radio hardware of the testbed is used. This hybrid solution combines the good visibility of simulators and the communication realism of real radio hardware. Another feature of *Emstar* is its hardware abstraction layer. It allows the developers to use the same application-code for simulation and for emulation without modification, which enables a fast transition between different simulation- and emulation modes. The operation mode that provides the best sensing-, computation-, and communication realism within *Emstar* is called *Portable–Array*. It is still a wired testbed but with its long serial cables it can be used also for outdoor experiments.

SeNeTs [BRGT05] is in many aspects similar to *Emstar*. Both run the same code on simulation and on the real node hardware and both incorporate an environment model. The main difference is that in *SeNeTs* the simulation part runs on distributed PCs, which improves scalability.

Services for real-world deployments

Deluge [HC04] is a data-dissemination protocol used for sending new code images over the air to deployed TinyOS sensor nodes. It uses the local memory on the nodes for caching the received images. A disseminated buggy code image could render a network unusable. This problem can be addressed with a *golden image* in combination with a watchdog-timer [DHJ+06]. The golden image is a known-working program that resides on every node, preferably on a write-protected memory section. Once an unrecoverable state is reached the watchdog-timer fires and the bootloader loads the golden image, which reestablishes the operability of the network.

Marionette [WTT+06] is an embedded RPC service for TinyOS programs. With some simple annotations, the compiler adds hooks into the code which allow a developer at run-time to remotely call functions and read or write variables. The main cost of using Marionette is that each interaction with a node requires network communication. Sharing the wireless channel with the application could adversely affect the behavior of the network algorithm that is being developed or debugged.

5.2 Deployment Support Networks

The Deployment Support Network (DSN) is a tool for the development, debugging and monitoring of distributed wireless embedded systems in a realistic environment. The basic idea is to use a second wireless network consisting of so-called DSN-nodes that are directly attached to the target nodes.

The DSN provides a separate reliable wireless backbone network for the transport of debug and control information from and to the target-nodes. However, it is not only a replacement for the cables in wired testbeds but it also implements interactive debugging services such as remote reprogramming, RPC and data/event-logging.

5.2.1 DSN–Architecture

5.2.1.1 Overview

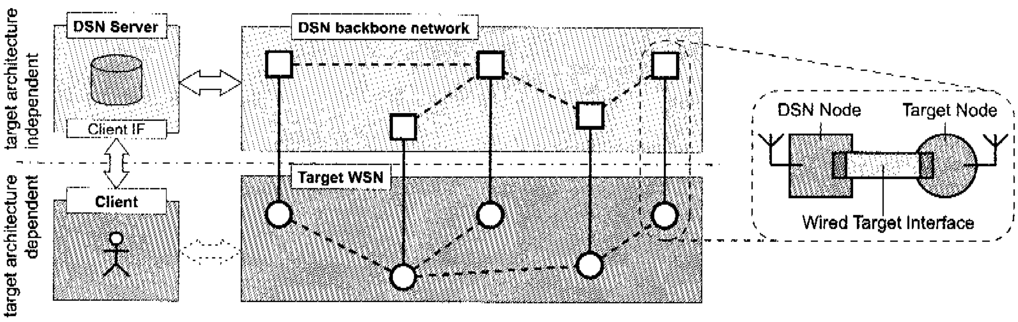


Fig. 46: Conceptual view of a DSN-system with five DSN-node/target-node pairs.

Figure 46 shows an overview of the different parts in a DSN-system. On the right hand side is the *DSN-node/target-node pair* that is connected via a short cable, referred to as the *wired target interface*. DSN-nodes are battery-operated wireless nodes with a microcontroller and a radio-module, similar to the target-nodes.

In the center of the figure, there is a conceptual view of the DSN with the two separate wireless networks: the one of the DSN-nodes and the one of the target-nodes. The network of the DSN-nodes is an automatically formed and maintained multi-hop backbone network, that is optimized for connectivity, reliability and robustness.

The *DSN-server* is connected with the DSN-backbone-network and provides the client interface, over which the client can communicate and use the implemented DSN-services. The *client* is a target-specific application or script. The information flow goes from the client over the DSN-server to the DSN-nodes and finally to the target nodes and vice versa. The DSN-server decouples the client from the target WSN both in time and space. In particular, data from the target nodes are stored in a database and can be requested anytime, and commands can be scheduled on the DSN-nodes. Separation in space is given through the client interface that allows for IP-based remote access.

5.2.1.2 Target-Architecture-Independent Services

A key feature of the DSN-system is the clear separation of the target-system and the DSN-services. As a result, the DSN can be used for the development and testing of different target-architectures. The DSN-services are target-architecture-independent. Only the *wired target interface* and a small part of the software on the DSN-nodes to control it must be adapted. However, this adaptation is typically a matter of I/O configuration which is completed fast.

5.2.1.3 Client Interface

The DSN-server provides a flexible RPC user interface for the DSN-services. A developer can write his own client scripts and test applications. The client virtually communicates over the DSN with the WSN application on the target-nodes.

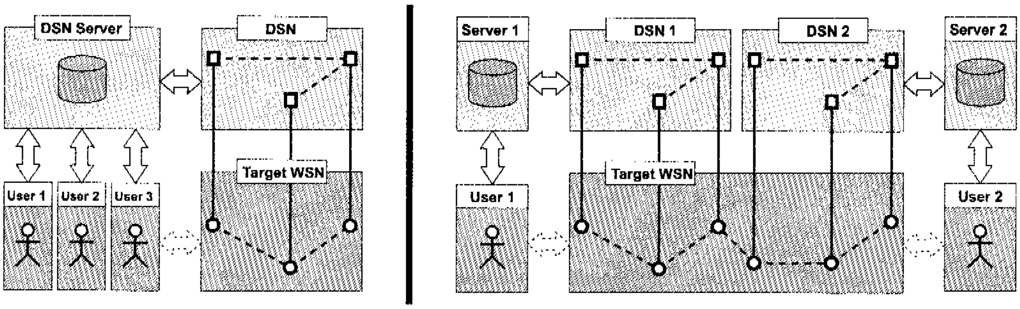


Fig. 47: DSN multi-user and multi-network

The DSN supports multiple users. Figure 47 shows two examples. Multiple users can connect to one DSN-server. Different access privileges can be assigned to users. For example this allows for read-only access or for privileged users that are permitted to reprogram the target nodes or to reconfigure the DSN. The second example shows two separate DSN-networks, each with its own server. By this means, two developers can work independently in the same location without interfering each other. Additionally this setup balances the load onto two networks, i.e. yielding a better performance. The separation is achieved by a unique DSN-network-ID that is checked on the DSN connection setup.

5.2.2 DSN-Services

In this section we describe the debugging services that are provided by the DSN. Each service has a part which is implemented on the DSN-server and a part that is implemented on the DSN-nodes.

5.2.2.1 Data- and Event-Logging

Probably the most important service of the DSN is the data- and event-logging. It gives the developers insight into the state of the target nodes. The basic concept is as follows: The target-nodes write logging-strings to the wired target-interface (by using e.g. printf-like statements for writing to a debug-UART). The DSN-node receives the log-string, annotates it with a time-stamp and stores it in a local logfile. On the other side, the DSN-server has a logging database, where it collects the log-messages from all the DSN-nodes. For that purpose there are two mechanisms: In *pull-mode*, the DSN-server requests the log-messages, where in *push-mode*, the DSN-nodes sends the log-messages proactively to the server (see Figures 48 and 49). Finally, the user can query the database to access the log-messages from all target-nodes.

String-based messages are very convenient for debugging and monitoring sensor network applications. They can be transmitted via a serial two-wire cable to the DSN-node with only little overhead on the target-nodes. The majority of the work is done on the DSN-nodes: They run a time-synchronization protocol for accurate time-stamping and they are responsible for transmitting the messages to the server. However, for certain experiments, even the overhead of writing short log-messages to a serial interface is not acceptable. Therefore, there is a second mechanism which uses I/O and external interrupt-lines to trigger an event. Events are, similar to the log-strings, time-stamped and cached in the logfile.

The caching of messages on the DSN-nodes is an important feature. It allows for a delayed transmission to the server, which is necessary for reducing the interference. The transmission can for example be scheduled by the server (pull-mode) or it is delayed until an experiment has finished. It even allows the DSN-nodes to disconnect entirely from the DSN for a certain time and then after reconnection to send all cached log-messages.

For the sake of platform-independence, the content of the log-messages is generally neither parsed by the DSN-nodes nor the DSN-server. This is the responsibility of the user application written by the developer who knows how to interpret the format. However, the DSN-nodes can optionally classify the messages into classes such as *Errors*, *Warnings* and *Debug*. This can be useful if one wishes to directly receive critical error-messages using the push-mode, while the bulky rest of the debugging messages can be pulled after the experiment.

5.2.2.2 Commands

The counterparts of the log-messages are the commands. With this service, string-based messages can be sent from the client to the target-nodes. There are two types of commands: *instant commands* that are executed immediately and *timed commands* that are schedulable. A destination-identifier that is given as a parameter, lets the client select either a single node or all nodes. Once the command is delivered to the DSN-server, it is transmitted immediately to the selected DSN-nodes. Then, in the case of an instant command, the message-string is sent over the wired target interface to the target-nodes. For the timed commands, a timer on the DSN-node is started which delays the delivery of the message. Again, the content of the message-string is not specified. It can be binary data or command-strings that are interpreted on the target-nodes.

Together with the data-logging, this service can be applied for the emulation of interactive terminal sessions with the target-nodes. This service sends commands to the nodes while the replies are sent back as log-messages to the user. In other words, this is a remote procedure call (RPC) that goes over the backbone network of the DSN. The wireless multi-hop network introduces a considerably larger delay than direct wired connections. However, even with a few seconds it is still acceptable for human interaction.

Timed commands are necessary if messages should be delivered at the same time to multiple target-nodes. The accuracy of time-synchronization of the DSN-nodes is orders of magnitude higher than the time-of-arrival of broadcasted messages. In addition this service can be used to upload a script with a set commands that will get executed on the specified time.

In addition to the described commands, there is an additional command for the wired target interface which lets the developer control the target-architecture specific functions such as switching on/off the target power, reading the target voltage, and controlling custom I/O lines.

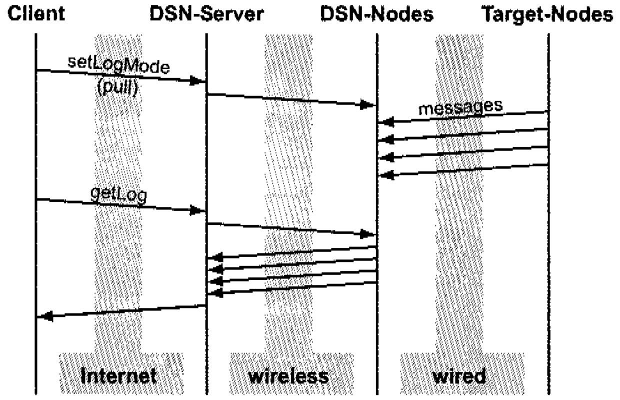


Fig. 48: Data- and event-logging in *pull-mode.*

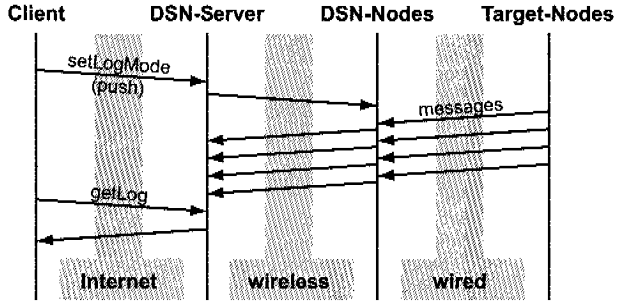


Fig. 49: Data- and event-logging in *push-mode.*

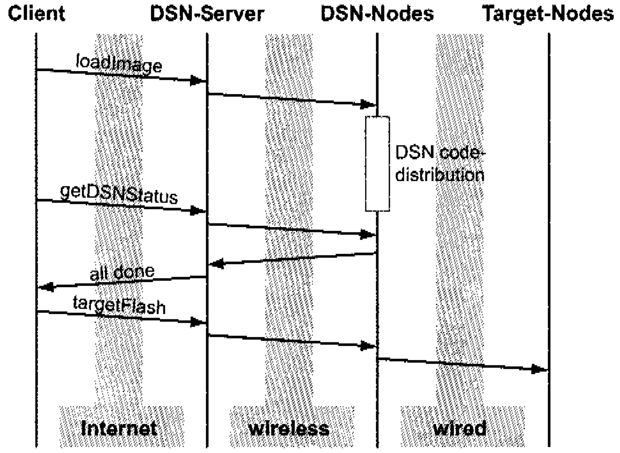


Fig. 50: Remote reprogramming with code distribution.

5.2.2.3 Remote Reprogramming

The DSN has a convenient remote-reprogramming service. The developer uploads a code image for the target-nodes to the server, which forwards it to the first DSN-node. The DSN-nodes have an extra memory section that is large enough to store a complete code image. They run a data-dissemination protocol to distribute the code image to all nodes over the backbone network. The nodes periodically send status-information to the direct neighbors including a version number and the type of the image. By doing so, also newly joined nodes with an old version get updated.

At any time, the developer can monitor the progress of the data dissemination. Once all DSN-nodes have received the code image, he can, with an additional command, select a set of DSN-nodes for the reprogramming of the target-nodes (see Figure 50). The DSN-node is connected to the programming-port of the target-node through the wired target interface. This programming connection and its software driver on the DSN-nodes is one of the few parts of the DSN-system that is architecture-dependent. It must be adapted if target-nodes with a new processor type are used.

5.2.2.4 DSN configuration and DSN status

The DSN is configurable. The user can set different operation-modes and parameters both at setup-time and at run-time. One such mode is the *low-power/low-interference mode.* When this mode is set, the DSN-nodes switch off their radio for a given time-interval. This might be important if the radio of the DSN and the one of the target-system interfere with each other. If this is the case, the DSN radio should be switched off during the experiment. As this mode is also very energy-efficient, it could be set whenever the DSN remains unused for a known time, e.g. during the night. This will significantly increase the life-time because only a timer on the microcontroller of the DSN-nodes needs to be powered.

The DSN further provides the developer with the possibility to gain information about the state of the DSN. In particular, the following information is provided:

- a *list of connected DSN-nodes* with a *last-seen* timestamp,
- a *location-identifier*,
- the *connectivity information* of the DSN-nodes,
- the *versions and types* of the stored code images, and
- the battery voltages of the DSN-nodes

The location-identifier is a string that is stored on every DSN-node containing e.g. the coordinates for positioning. It can be set via the user-interface when the DSN is deployed.

The gathering of the DSN-status requires bandwith on the DSN backbone network. To minimize this overhead, the DSN-server only fetches the information from the DSN-nodes when it is explicitly requested by the client. Otherwise it provides a cached version of the DSN-status.

5.2.3 Test Automation

It is often not enough to have an interactive terminal session to the nodes. There is a need for automation and scripting support for the following reasons: (a) Experiments have to be repeated many times, either for statistical validity or to explore different parameter settings. (b) The execution of an experiment has to be delayed, e.g. since interference caused by human activity is minimized during nighttime. (c) Experiments last longer than an operator can assist.

One possibility to automate tests is to send once a set of timed commands to the DSN-nodes (see section 5.2.2.2). However, a more sophisticated method for test automation is to use scripts that interact with the DSN-server. This has the advantage that a script can evaluate the state of the targets during test execution and adapt its further actions. For example, only when a particular message from node A is received, a command to node B is sent.

The above described DSN-services are accessible as RPC functions and can therefore be called easily from scripts. Table 9 shows some functions for the different parts.

```
test-setup:
    loadImage(type, version, code image)
    targetFlash(selected-nodes)
    dsnConfig([selected-nodes], property, value)
    setLogMode(selected-nodes, class, push|pull)
test-execution:
    instantCommand(selected-nodes, command)
    timedCommand(selected-nodes, command, time)
result gathering:
    getDSNStatus()
    getLog(filter)
```

Tab. 9: Pseudo-syntax of the RPC functions

5.3 Realization

In the previous section, we described the general concept and methodology of the DSN. In this section we present our implementation. Figure 51 shows an overview of the technologies used in our implementation. See also [BTn] for details.

For the DSN-nodes we use the BTnodes rev3 [BDH+04]. This platform has proven to be a good choice since it has a relatively large memory and a robust radio.

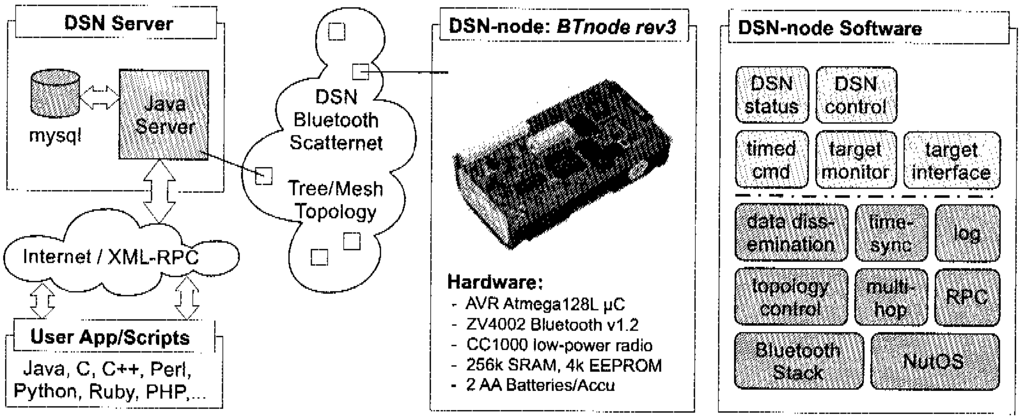


Fig. 51: Technology overview of the DSN implementation on the BTnodes rev3.

5.3.1 Bluetooth Scatternets

Bluetooth is not often seen on sensor-networks, due to its high energy-consumption (BTnode: 100 mW). However, it has a number of properties that the traditional sensor-network radios do not have and which are very important for the DSN backbone network. Bluetooth was initially designed as a cable-replacement. It provides very robust connections. Using a spread-spectrum frequency-hopping scheme, it is resilient against interference and has a high spatial capacity. Robustness and spatial capacity are mission-critical for the DSN.

Using Bluetooth for the DSN has further the benefit that it is potentially accessible from PDAs and mobile phones. Although this not utilized in our current implementation, it opens interesting new usage scenarios for the future.

We use the BTnut system software on the BTnodes which comes with an embedded Bluetooth stack. For the automatic formation and maintenance of a connected Bluetooth scatternet, we use the topology control algorithms presented in Chapter 4. They are adaptive algorithms, i.e. taking network changes due to link-losses and leaving or joining nodes into account.

5.3.2 Wired Target Interface

The only part of the DSN-node software that must be adapted for new target architectures is the *target interface.* Common to most platforms is that data transport is possible through a serial RS232-like connection. The BTnode provides a hardware UART for this purpose. Porting this interface to similar platforms consist of adapting the bitrate and configuring flow control.

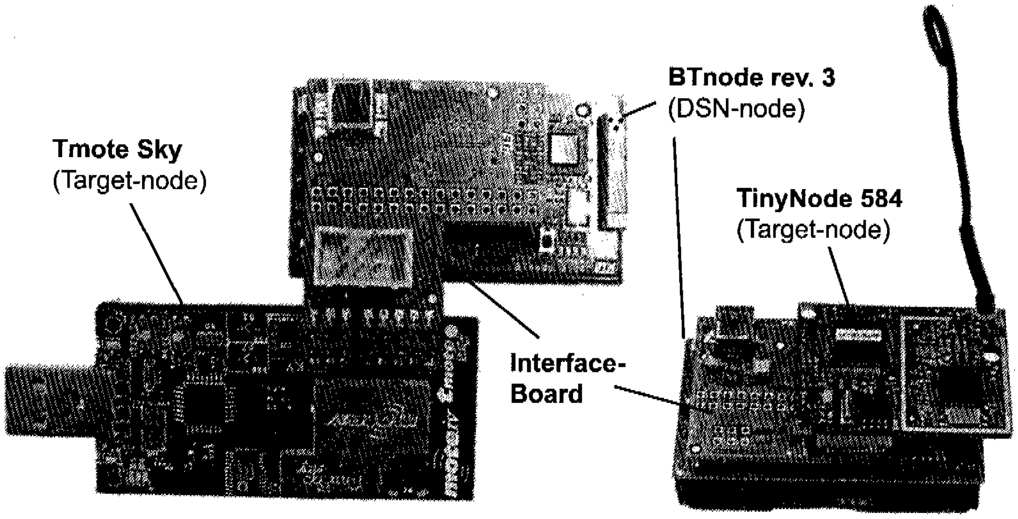


Fig. 52: Two example of realized target interfaces: DSN-node–target-node pairs are shown with the BTnode rev.3 and the Tmote Sky (left) and the Shockfish TinyNode 584 (right).

More problematic is the programming connection, since different platforms have quite different solutions. Some target-architectures provide direct access to the programming port (ISP). For this case the target-interface must execute the ISP protocol of the appropriate microcontroller type. We have this currently implemented for the AVR and the MSP430 microcontroller family. Some other target-architectures use the same serial port both for data transport and programming. The appropriate control signals for the multiplexing must then be issued by a custom function of the target interface on the DSN-node.

A third programming method is applied on the *Tmote Sky* target: We had to program the targets with a custom bootloader that is able to receive the code image over the external pins (instead of the USB-connector). Figure 52 shows two different DSN-node - target-node pairs for which our implementation supports the general DSN-services. We further implemented an interface for the BTnode- and the A80 target. The A80 is a radio and processing module developed by Siemens that is used in wireless fire detectors. For the BTnode- and the A80 target, we added additional target-monitoring functions such as target-power sensing and control to the wired target interface.

5.3.3 Client Interface

All DSN-services are accessible as RPC functions. We use XML-RPC, since it is platform independent and there exist API-libraries for a large number of programming- and scripting languages. The application or script which uses the DSN-services is target-architecture dependent and must therefore be written by the user. The script in Figure 53 demonstrates how simple automation and experimental setups can be written. In this example, a code image is uploaded to the server, the data dissemination is started, and then the targets are programmed. The DSN does not perform a version check of the targets since this is dependent on the target-application. This must therefore be a part of the client script. In the example it is assumed that the targets write out a boot-message, such as `"Version:X375"`. The script uses both a time-string and a text-filter-string to query the server for the corresponding log-messages.

5.3.4 Performance Evaluation

The performance of the implemented S-XTC algorithm on the BTnodes is mostly limited by the packet processing soft- and hardware. In fact, the microcontroller is too slow for the packet processing at full Bluetooth-speed. Incoming packets are stored in a receive buffer. If the arrival rate of packets is higher than the processing rate for a certain time, packets are dropped due to the limited capacity of the receive buffer. This affects the performance of the DSN in several ways: (a) log-messages in push-mode might get lost, (b) log-messages in pull-mode might get lost, (c) commands might get lost, and (d) data-dissemination packets might get lost. The probability of these cases increases with the amount of traffic on the DSN backbone network. For many scenarios the user can control what and when data is sent on the DSN. He can e.g. wait for the completion of the data-dissemination before he starts pulling messages. In general, cases (b)-(d) are not critical, as they can be resolved with retransmission. However, in a scenario, where all nodes periodically generate log-messages that are pushed simultaneously to the server, the log-messages can not be retransmitted. So for case (a), the user wants to know the transport-capacity of the DSN, such that he can adjust the parameters of the setup.

In Figure 54, we show the measured yield of correctly received log-messages at the server. We varied the message-generation rate from 0.5 to 4 packets per node per second and the DSN size from 10 to 25 nodes.

```
user script example (Perl)

require
RPC::XML::Client;

# creates an xml-rpc connection to the dsn-server
  $serverURL='http://tec-pc-btnode.ethz.ch:8888';
  $client = RPC::XML::Client->new($serverURL);

# sends the code image to the dsn-server with xml-rpc
  $filename = 'experiment2.hex'; $handle = fopen($filename, 'r');
  $req = RPC::XML::request->new('dsnService.uploadFile', $filename,
                               RPC::XML::base64->new($handle));
  $resp = $client->send_request($req);

# initiates the data-dissemination on the dsn-nodes
  $type = 1;       # code image is for target nodes
  $resp = $client->send_request('dsnService.loadFile', $filename, $type);

# wrapped function that uses 'dsnService.getDSNstatus' to wait until
# all targets have received the code image
  waitDataDisseminationComplete($filename);

# programm the targets
  $flashtime = $client->send_request('dsnService.getServerTime');
  $resp = $client->send_request('dsnService.targetFlash', 'all');
  sleep(5);

# collects the target-versions sent as boot-message by targets
  $resp = $client->send_request('dsnLog.getLog', 'all', 31, 18, 'Version: X',
                               $flashtime, '');
  @versions = ();
  for $entry (@{$resp}){
    $entry{'LogText'} =~ m/^Version: X(\d+)/;
    push(@versions, {'node' => $entry('DSNID'), 'version' => $1});
  }
```

Fig. 53: User script example in Perl

Each message carries 86 bytes payload. We left this value constant, because sending the double amount of data would result in sending two packets, which is the same as doubling the message rate. The measurements are performed on random topologies that were generated by the integrated tree topology algorithm. We observed slightly different results for different topologies, but all with the same characteristic: Starting with slow message rates, all packets are received correctly. However, there is a certain rate, from which on the yield decreases very quickly. This cut-off point is between 0.5 and 1 messages per second for 25 and 20 nodes, between 1.5 and 2 for 15 nodes, and between 2.5 and 3 for 10 nodes. Thus, if for this streaming scenario a developer needs all pushed log-messages, he must set the message-rate or the DSN size below this cut-off point.

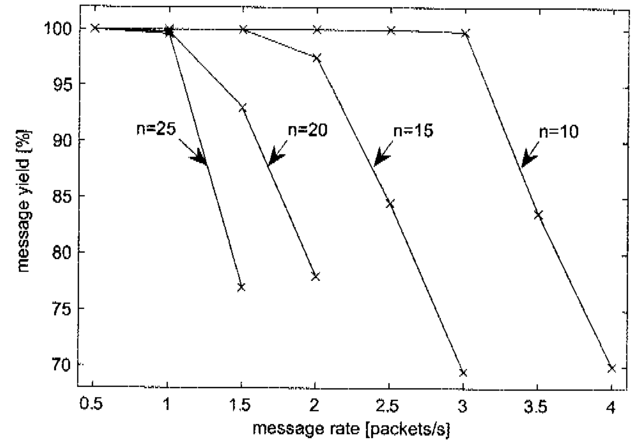


Fig. 54: Yield of correctly received log-messages that are pushed periodically to the server for different message-rates and different sizes of the DSN. For the measurement each node sent 100 log-messages with 86 bytes payload.

5.4 Case-Study: Link Characterization in Buildings

Some wireless applications in buildings require highly reliable communication. A thorough understanding of radio propagation characteristics in buildings is necessary for a system design that can provide this reliability. Engineers of Siemens Building Technologies use a BTnode-based DSN system to measure and evaluate link characteristics of wireless fire-detectors. To this purpose, the DSN system remotely controls the target nodes and collects measurement data. In the following, the measurement setup is described and the *type* of results that can be obtained is presented. The purpose of the case study is to proof the concept of the DSN system, therefore the obtained data is not discussed in detail.

5.4.1 Experiments

The measurement setup consists of up to 30 target nodes, each connected to a DSN node. Nodes are placed at exactly the locations in the building, where the future fire-detectors will be deployed. (see Figure 55). We measure signal strength (RSSI) and frame error rates for every link between any two target nodes. Additionally, noise levels and bit error rates are evaluated. One target node is sending a fixed number of test frames while all the others are listening and recording errors by comparing the received frame to a reference-frame. Two messages are generated per second. During and after the reception of every frame, the RSSI is recorded

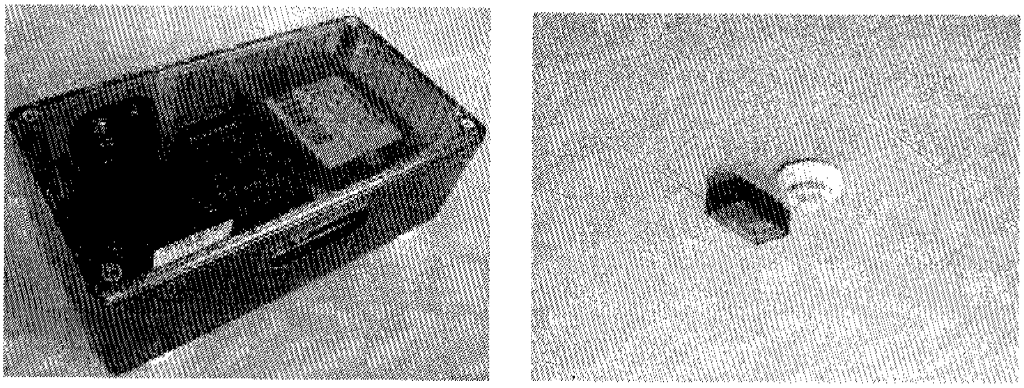


Fig. 55: Siemens "Blue Box" with BTnode, A80 target node and batteries. The Adapter Board acts as a connecting cable. The Box is placed close to existing fire-detectors for a realistic setup.

in order to provide information about the signal and noise levels.

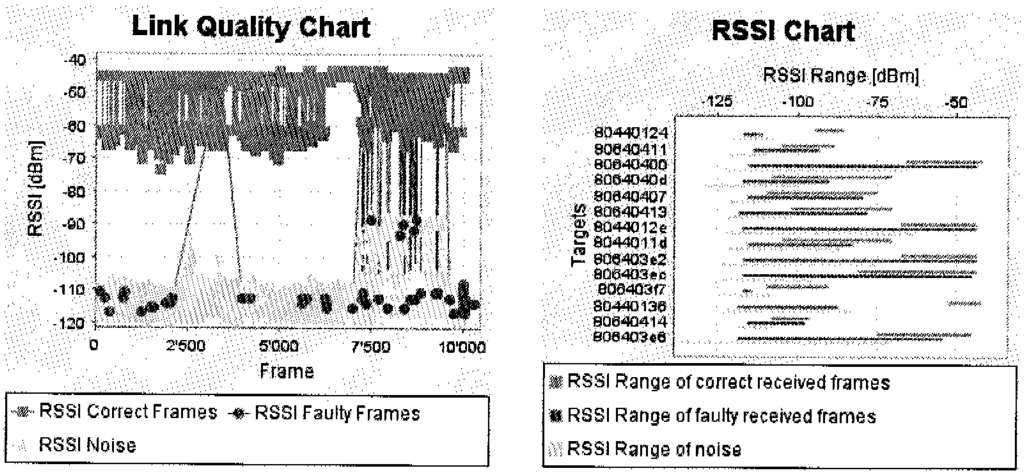


Fig. 56: Detailed mode (left) and summary mode (right) of the RSSI measurements.

In the *detailed mode*, receiving target nodes create a message after every frame reception. Figure 56 shows on the left the data collected by one target node in detailed mode. In *summary mode*, receiving target nodes only create a message after a complete sequence of frames has been received. Figure 56 shows on the right the data collected by 14 nodes in summary mode. In summary mode, the amount of data is significantly reduced compared to detailed mode. This allowed us to concurrently evaluate all 30 target nodes in the test setup. On the other hand, only detailed mode (with maximally 10 nodes, see also Figure 54) allowed us to analyze the temporal properties of collected data. E.g. Figure 56 shows that frames with low signal strength do not occur at random times, but are concentrated towards the end of the experiment. Thus channel properties seem to vary over time.

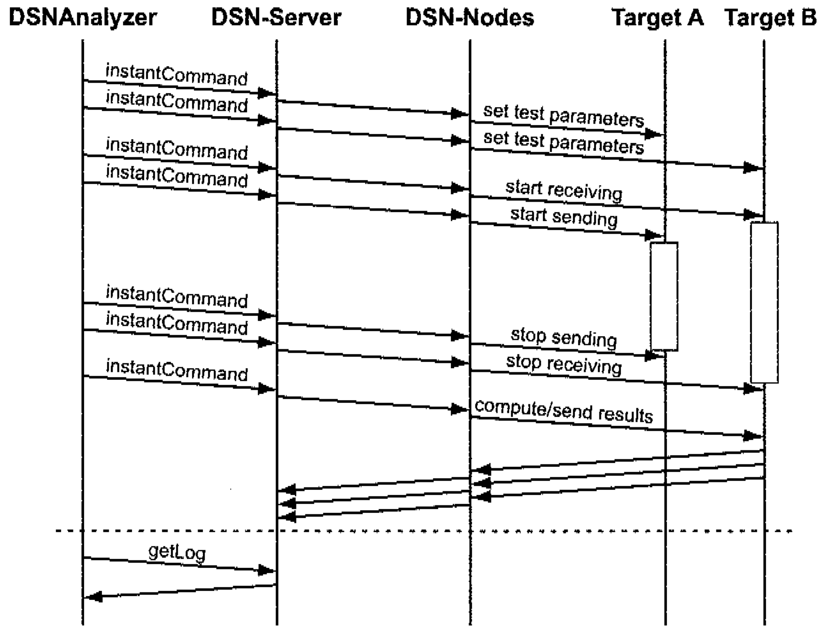


Fig. 57: The automated execution of a simple test from the *DSNAnalyzer* which is a client of the DSN.

The procedure described above provides data for the links between one sending target and all other targets. Test automation is used to repeat this procedure with different senders such that finally the links between any two nodes in the target system are evaluated. Test automation is also used to repeat tests with different WSN-system parameters like e.g. transmit power. Finally tests are executed during day- and nighttime to observe the influence of human interference. In this case a Java application acts as the user of the DSN-system (see Figure 46). The interaction of this application with the DSN system is illustrated in Figure 57.

5.5 Summary

We have presented the Deployment Support Network. It is a new methodology to design and test sensor-network applications in a realistic environment. Existing solutions fail at providing at the same time both visibility and the high quality information from real deployments.

The DSN is wireless, which is the key difference to existing emulation testbeds. The deployment of DSN-node/target-node pairs is much easier than handling hundreds of meters of cables. This means that the positions of the nodes and thus the density of the network can be chosen and adjusted quickly according to the application requirements and is no longer dictated by the testbed setup.

However, using wireless ad-hoc communication instead of cabled infrastructure introduces also new limitations. One is the limited range of the DSN radio. If the range of the targets radio is larger than the one of the DSN-nodes and if a sparse deployment with maximal distances between the nodes is to be tested, additional DSN-nodes have to be inserted that act as repeaters. Another limitation is obviously the lower throughput for debugging and control information. A researcher must be aware of this and choose the rate of generated pushed messages accordingly or change to pull mode if possible. In our implementation Bluetooth provides the necessary robustness and reliability needed for the DSN. With its high spatial capacity it allows not only for large deployments, but also for very dense ones.

Compared to existing services for real-world deployments such as Deluge and Marionette, the DSN is different in the sense that the services run on separate hardware and not on the target-nodes itself. This solution causes less interference since debugging services and the sensor-network application are clearly separated and do not share the same computing and radio resources. The resource demand of the DSN-services is different from the resource demand of the target-application which asks for different architectures. If in an application scenario the nodes only have to transmit a few bits once every 10 minutes with best effort, the developer would choose an appropriate low-power/low-bandwidth technology. Running the DSN-services over such a network is not feasible. Another approach is over-engineering. One could use more powerful nodes for the sake of better visibility and flexibility during development. Running a data-dissemination service on the target-nodes would require additional memory that is large enough for a whole code image. Expensive extra memory that is only used for development is no feasible option for industrial products.

During development and test, the DSN-nodes execute the services on dedicated optimized hardware. After that, they can be detached from the target-nodes. Since the services are implemented on the DSN they can be used for different target architectures and independently of their operating system.

The BTnode based DSN system has proved to be very useful for SBT's development teams. The following advantages were most relevant to them:

- The simple interface between DSN and target nodes makes it possible to work with existing target platforms. Alternative systems require specific hard or software on the target side.
- The DSN-node/target-node pairs are completely wireless and thus

can be deployed quickly and even in inaccessible locations. This is important in their use-case since they are collecting data from a wide range of buildings, some of them in use by their customers, which excludes wired installations.

6

Conclusions

In this chapter, we summarise our contributions and discuss potential extensions of the work for future research.

6.1 Summary of Contributions

With the work presented in this thesis, we contribute towards an increased quality of the design of distributed embedded systems. Our research on validation strategies addresses the challenging problems that arise from the peculiarities of distributed embedded systems such as the unreliable wireless communication, the limited resources and the problem of accessing state information on nodes:

- **Estimation of a wearable system**
  As an instance of a distributed embedded system, a wearable system consists of multiple modules with sensing-, actor-, and computing devices, that are distributed over the human body. In Chapter 2 we presented the modeling and performance estimation of such systems for multiple objectives. The main contributions of this chapter to the thesis are the analytic models for the estimation-based validation. In particular, we combined models of the usage scenario with models of computing- and wired- and wireless communication devices.

- **Virtualized execution of HW-Tasks**
  We propose in Chapter 3 the virtualized execution in distributed embedded systems when applications are executed on reconfigurable

hardware. Applications are partitioned into a number of hardware task that are executed following the semantics of the process network model. This has a number of advantages for design and validation: Tasks can be validated and deployed separately and can be reused multiple times.

- **Simulation and testing of distributed algorithms**
  We discuss in Chapter 4 distributed algorithms on the example of the topology control problem. Starting from an existing algorithm description, we validated the practicability on real sensor nodes. The combination of simulation, measurements and a implementation-driven approach allowed us to identify assumptions that are impractical but that appear nevertheless often in descriptions of distributed algorithms. We further propose additional heuristics that account for the imperfection that appear in a real-world scenario. We tested the improved topology control algorithm on the BTnode platform in a realistic testbed with up to 50 nodes.

- **Deployment Support Networks**
  A Wireless Sensor Network is a distributed embedded system that is very difficult to validate. The key challenge is to capture the complex physical phenomena that appear in a real-world scenario in the system testing. In Chapter 5 we contribute to the thesis by proposing the Deployment Support Network, a new validation methodology for the development and test of WSNs. It is a platform-independent toolkit that allows a developer to test applications on the real execution platform and on the real deployment with minimal modification and interference. The Deployment Support Network provides a number of services for monitoring, data-logging and control which can be used in automated test cases. To this end, the proposed methodology is used by several academic projects as well as by an industrial partner for the testing of new distributed embedded systems.

6.2 Future Directions

In the following, we will briefly outline possible directions for further research:

The proposed Deployment Support Network provides services for the observation and control of deployed nodes under test. We further described how this services can be used in an automated test scenario. However, these are only the first steps towards a generic automated supervision system. We envision a precisely defined methodology for executing system tests for wireless sensor networks which includes the definition of test interfaces and the generation of test data. With this thesis as a basis, we believe that further confirmed test methods from the software engineering domain can be adapted to wireless sensor networks. One example is test driven design and regression testing. Using a component-based design, small individual parts of the system software can be specified, implemented and validated independently. By consequently writing test-cases for every newly implemented component, a developer can perform regression testing in order to validate that no new errors are introduced in other components. With such an approach we would foster researchers to start earlier with testing on the actual platform and not just at the very end of the design flow. Consequently, this would bridge the gap between the abstract system design world and the real world and increase the quality of the design.

Bibliography

[ABD⁺04] U. Anliker, J. Beutel, M. Dyer, R. Enzler, P. Lukowicz, L. Thiele, and G. Tröster. A systematic approach to the design of distributed wearable systems. *IEEE Transactions on Computers*, 53(8):1017–1033, August 2004.

[ALE02] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35(2):59–67, February 2002.

[ASI⁺98] A. Abnous, K. Seno, Y. Ichikawa, M. Wan, and Jan Rabaey. Evaluation of a low-power reconfigurable dsp architecture. In *Proceedings of the 5th Reconfigurable Architectures Workshop (RAW)*, number 1388 in Lecture Notes in Computer Science, pages 55–60. Springer, Berlin, 1998.

[Atm03] Atmel Corp. Field programmable system level integrated circuit (FPSLIC), 2003.

[BBD⁺07] J. Beutel, Ph. Blum, M. Dyer, C. Moser, M. Yücel, and Ph. Stadelmann. *BTnode Programming – An Introduction to BTnut Applications*, rev. 1.5 edition, 2007.

[BBMP04] S. Basagni, R. Bruno, G. Mambrini, and C. Petrioli. Comparative performance evaluation of scatternet formation protocols for networks of Bluetooth devices. *Wireless Networks*, 10(2):197–213, March 2004.

[BD01] G. Brebner and O. Diessel. Chip-based reconfigurable task management. In *Proceedings of the 11th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 182–191. Springer, 2001.

[BDH⁺04] J. Beutel, M. Dyer, M. Hinz, L. Meier, and M. Ringwald. Next-generation prototyping of sensor networks. In *Proc. 2nd ACM Conf. Embedded Networked Sensor Systems (SenSys*

2004), pages 291–292. ACM Press, New York, November 2004.

[BDMT05] J. Beutel, M. Dyer, L. Meier, and L. Thiele. Scalable topology control for deployment-support networks. In *Proc. 4th Int'l Conf. Information Processing in Sensor Networks (IPSN '05)*, pages 359–363. IEEE, Piscataway, NJ, April 2005.

[Beu05] J. Beutel. *Design and Deployment of Wireless Networked Embedded Systems.* PhD thesis, Diss. ETH No. 16204, ETH Zurich, 2005.

[BHW99] C. Baber, D. J. Haniff, and S. I. Woolley. Contrasting paradigms for the development of wearable computers. *IBM Systems J.*, 38(4):551–565, 1999.

[BL03] Douglas M. Blough and Mauro Leoncini. The k-neigh protocol for symmetric topology control in ad hoc networks. In *Proc. of the 4th ACM international symposium on Mobile ad hoc networking & computing (MobiHoc)*, 2003.

[Bre96] Gordon Brebner. A virtual hardware operating system for the xilinx xc6200. In *Proceedings of the 6th International Conference on Field-Programmable Logic and Applications (FPL)*, Lecture Notes in Computer Science, pages 327–336. Springer, Berlin, 1996.

[BRGT05] J. Blumenthal, F. Reichenbach, F. Golatowski, and D. Timmermann. Controlling wireless sensor networks using senets and envisense. In *Proc. 3rd IEEE International Conference on Industrial Informatics (INDIN)*, pages 262 – 267, 10-12 Aug 2005.

[BTn] BTnodes. A distributed environment for prototyping ad hoc networks. http://www.btnode.ethz.ch.

[BTT98] T. Blickle, J. Teich, and L. Thiele. System-level synthesis using evolutionary algorithms. *Design Automation for Embedded Systems*, 3(1):23–58, January 1998.

[CB96] Z. Cvetanovic and D. Bhandarkar. Performance characterization of the Alpha 21164 microprocessor using TP and SPEC workloads. In *Proc. 2nd IEEE Symp. on High-Performance Computer Architecture (HPCA)*, pages 270–280, 1996.

[CCH+00] E. Caspi, M. Chu, R. Huang, J. Yeh, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfigurable execution (SCORE). In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL)*, volume 1896 of *Lecture Notes in Computer Science*, pages 605–614. Springer-Verlag, 2000.

[DBM05] M. Dyer, J. Beutel, and L. Meier. Deployment support for wireless sensor networks. In *4. GI/ITG KuVS Fachgespraech Drahtlose Sensornetze*, pages 25–28, ETH Zurich, March 2005. referred workshop.

[DBT07a] M. Dyer, J. Beutel, and L. Thiele. S-xtc: A signal-strength based topology control algorithm for sensor networks. In *Proc. of the 8th Intl. Symposium on Autonomous Decentralized Systems (ISADS)*, pages 508–515. IEEE CS Press, Los Alamitos, CA, March 2007.

[DBT+07b] M. Dyer, J. Beutel, L. Thiele, T. Kalt, P. Oehen, K. Martin, and Ph. Blum. Deployment support network - a toolkit for the development of wsns. In *Proceedings of the 4th European Conference on Wireless Sensor Networks*, pages 195–211, January 2007.

[DHJ+06] P. Dutta, J. Hui, J. Jeong, S. Kim, C. Sharp, J. Taneja, G. Tolle, K. Whitehouse, and D. Culler. Trio: enabling sustainable and scalable outdoor wireless sensor network deployments. In *Proc. of the fifth international conference on Information processing in sensor networks (IPSN)*, pages 407–415. ACM Press, New York, 2006.

[dKSvdW+00] E. A. de Kock, W. J. M. Smits, P. van der Wolf, J.-Y. Brunel, W. M. Kruijtzer, P. Lieverse, K. A. Vissers, and G. Essink. Yapi: application modeling for signal processing systems. In *Proceedings of the 37th conference on Design automation (DAC)*, pages 402–405, New York, NY, USA, 2000. ACM Press.

[DPP02] M. Dyer, C. Plessl, and M. Platzner. Partially reconfigurable cores for xilinx virtex. In *Proceedings of the 12th International Conference on Field-Programmable Logic and Applications (FPL)*, volume 2438 of *Lecture Notes in Computer Science*, pages 292–301, Montpellier (La Grande-Motte), France, September 2002. Springer, Berlin.

[DPT04] M. Dyer, M. Platzner, and L. Thiele. Efficient execution of process networks on a reconfigurable hardware virtual machine. In *Proc. 12th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 342–344, Napa, CA, USA, April 2004. IEEE CS Press, Los Alamitos, CA.

[DTH02] O. Dousse, P. Thiran, and M. Hasler. Connectivity in ad-hoc and hybrid networks. In *Proc. of the 21th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2002.

[EGE04] J. Elson, L. Girod, and D. Estrin. Emstar: development with high system visibility. *Wireless Communications, IEEE [see also IEEE Personal Communications]*, 11(6):70–77, 2004.

[Enz04] R. Enzler. *Architectural trade-offs in dynamically reconfigurable processors.* PhD thesis, Diss. ETH No. 15423, ETH Zurich, 2004.

[ETZ00] M. Eisenring, L. Thiele, and E. Zitzler. Handling conflicting criteria in embedded system design. *IEEE Des. Test. Comput.*, 17(2):51–59, April-June 2000.

[GB03] M. Geilen and T. Basten. Requirements on the execution of kahn process networks. In *Lecture Notes in Computer Science, ESOP 2003*, volume 2618. Springer, 2003.

[GLS00] Steven A. Guccione, Delon Levi, and Prasanna Sundararajan. JBits: A Java-based Interface for Reconfigurable Computing. In *Proceedings of the 2nd Annual Military and Aerospace Applications of Programmable Devices and Technologies Conference (MAPLD)*, 2000.

[Gup95] R. K. Gupta. *Co-Synthesis of Hardware and Software for Digital Embedded Systems.* Kluwer Academic Publishers, August 1995.

[HC04] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM Press.

[HK02] T.A. Henzinger and C.M. Kirsch. The embedded machine: Predictable, portable real-time code. In *Proceedings of the*

ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2002.

[HL86] T. Hou and V. Li. Transmission range control in multihop packet radio networks. *IEEE Transactions on Communications*, 34(1):38–44, 1986.

[HL01] Edson L. Horta and John W. Lockwood. PARBIT: A Tool to Transform Bitfiles to Implement Partial Reconfiguration of Field Programmable Gate Arrays (FPGAs). Technical report, Department of Computer Science, Applied Research Lab, Washington University, Saint Louis, July 2001.

[HSE⁺00] Y. Ha, P. Schaumont, M. Engels, S. Vernalde, F. Potargent, L. Rijnders, and H. De Man. A hardware virtual machine for the networked reconfiguration. In *IEEE International Workshop on Rapid System Prototyping*, pages 194–199, 2000.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In J.L. Rosenfeld, editor, *Information Processing*. Ed. North-Holland Publishing Co., 1974.

[KAH⁺04] R. Kling, R. Adler, J. Huang, V. Hummel, and L. Nachman. Intel mote: Using Bluetooth in sensor networks. In *Proc. 2nd ACM Conf. Embedded Networked Sensor Systems (SenSys)*, page 318. ACM Press, New York, November 2004.

[KC98] I. Karkowski and H. Corporaal. Design space exploration algorithm for heterogeneous multi-processor embedded system design. In *Proc. 35th Design Automation Conf. (DAC)*, pages 82–87, 1998.

[KDR01] B. Kienhuis, E. Deprettere, and E. Rypkema. Compilation from matlab to process networks. In *Proceedings of the second international workshop on Compiler and Architecture Support for Embedded Systems (CASES)*, 2001.

[KKS01] M. Kourogi, T. Kurata, and K. Sakaue. A panorama-based method of personal positioning and orientation and its real-time application for wearable computers. In *Proc. 5th Int. Symp. on Wearable Computers (ISWC)*, pages 107–114, 2001.

[KL01] O. Kasten and M. Langheinrich. First experiences with Bluetooth in the Smart-It's distributed sensor network. In *Workshop on Ubiquitous Computing and Communication,* *Int'l Conf. Parallel Architectures and Compilation Techniques (PACT 2001)*, September 2001.

[KMW04] F. Kuhn, T. Moscibroda, and R. Wattenhofer. Initializing newly deployed ad hoc and sensor networks. In *Proc. 10th ACM/IEEE Ann. Int'l Conf. Mobile Computing and Networking (MobiCom 2004)*, pages 260–274, 2004.

[LAT+01] P. Lukowicz, U. Anliker, G. Tröster, S. Schwartz, and R. DeVaul. The WearARM modular, low-power computing core. *IEEE Micro*, 21:16–28, May/June 2001.

[LBV06] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture. In *Parallel and Distributed Processing Symposium20th International*, page 8 pp., April 2006.

[LCBK01] J. Liu, P. H. Chou, N. Bagherzadeh, and F. Kurdahi. A constraint-based application model and scheduling techniques for power-aware systems. In *Proc. 9th Int. Symp. on Hardware/Software Codesign (CODES)*, pages 153–158, 2001.

[LJS+02] P. Lukowicz, H. Junker, M. Stäger, T. von Büren, and G. Tröster. WearNET: A distributed multi-sensor system for context aware wearables. In *Proc. 4th Int. Conf. on Ubiquitous Computing (UbiComp)*, 2002.

[LK03] S. Lange and U. Kebschull. Virtual hardware byte code as a design platform for reconfigurable embedded systems. In *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE)*, pages 302–307, 2003.

[LLWC03] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proc. of the 1st int'l conference on Embedded networked sensor systems (SenSys)*, pages 126–137. ACM Press, New York, November 2003.

[LRD01] K. Lahiri, A. Raghunathan, and S. Dey. System-level performance analysis for designing on-chip communication architectures. *IEEE Trans. Computer-Aided Design*, 20(6):768–783, 2001.

[LSW+01] P.H.W. Leong, C.W. Sham, W.C. Wong, W.S. Yuen, and M.P. Leong. A Bitstream Reconfigurable FPGA Implementation of the WSAT Algorithm. *IEEE Transactions on VLSI Systems*, 9(1):197–201, February 2001.

[Man98] S. Mann. Wearable computing as means for personal empowerment. In *Proc. 3rd Int. Conf. on Wearable Computing (ICWC)*, May 1998.

[Mar03] P. Marwedel. *Embedded System Design*. Kulwer Academic Publishers, 2003.

[Mic94] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[MMF98] O. Mencer, M. Morf, and M. J. Flynn. Hardware software tri-design of encryption for mobile communication units. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, pages 12–15, 1998.

[MNC+03] J-Y. Mignolet, V. Nollet, P. Coene, D. Verkest, Vernalde S., and R. Lauwreins. Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable System-on-Chip. In *Proceedings of Design, Automation and Test in Europe (DATE)*, pages 986–991. IEEE Computer Society, March 2003.

[NBD06] L. Negri, J. Beutel, and M. Dyer. The power consumption of bluetooth scatternets. In *IEEE Consumer Communications and Networking Conference*, page to appear. IEEE, Piscataway, NJ, 2006.

[ns2] ns2. The network simulator - ns-2. Available via http://www.isi.edu/nsnam/ns/ (accessed July 2006).

[Par95] T.M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, Berkeley, University of California, 1995.

[PBC04] C. Petrioli, S. Basagni, and I. Chlamtac. BlueMesh: degree-constrained multi-hop scatternet formation for bluetooth networks. *Mob. Netw. Appl.*, 9(1):33–47, 2004.

[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, 2004.

[PEW+03] C. Plessl, R. Enzler, H. Walder, J. Beutel, M. Platzner, L. Thiele, and G. Tröster. The case for reconfigurable hardware in wearable computing. *Personal and Ubiquitous Computing*, 7(5):299–308, October 2003.

[PLS01] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *Proc. 7th ACM Int. Conf. on Mobile Computing and Networking (Mobicom)*, pages 251–259, 2001.

[PSC05] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *Proc. 4th International Symposium on Information Processing in Sensor Networks (IPSN)*, pages 364–369, 2005.

[RM99] V. Rodoplu and T.H. Meng. Minimum energy mobile wireless networks. *IEEE Journal on Selected Areas in Communications*, 17(8):1333–1344, August 1999.

[RM04] K. Römer and F. Mattern. The design space of wireless sensor networks. *IEEE Wireless Communications*, 11(6):54–61, December 2004.

[SC01] A. Sinha and A. P. Chandrakasan. JouleTrack - a web based tool for software energy profiling. In *Proc. 38th Design Automation Conf. (DAC)*, pages 220–225, 2001.

[SGVV02] G. Stitt, B. Grattan, J. Villarreal, and F. Vahid. Using on-chip configurable logic to reduce embedded system software energy. In *Proceedings of the 10th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 22–24, 2002.

[SHC+04] V. Shnayder, M. Hempstead, B. Chen, G. W. Allen, and M. 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 Press.

[SJR01] Satnam Singh and Phil James-Roxby. Lava and JBits: From HDL to Bitstream in Seconds. In *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines (FCCM)*, 2001.

[SK97] M. Stemm and R. Katz. Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Trans. Communications*, E80-B(8):1125–1131, August 1997.

[SLM00] H. Simmler, L. Levinson, and R. Männer. Multitasking on fpga coprocessors. In *Proceedings of the 10th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 121–130. Springer, 2000.

[SN97] R. Steinmetz and K. Nahrstedt. *Multimedia: Computing, Communications & Applications.* Prentice-Hall, 1997.

[SPMC04] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a sensor network expedition. In *Proceedings of the First European Workshop on Sensor Networks (EWSN)*, jan 2004.

[SRS00] A. Smailagic, D. Reilly, and D. P. Siewiorek. A system-level approach to power/performance optimization in wearable computers. In *Proc. IEEE Computer Society Workshop on VLSI (WVLSI)*, pages 15–20, 2000.

[SS99] A. Smailagic and D. Siewiorek. System level design as applied to CMU wearable computers. *J. VLSI Signal Processing Systems for Signal, Image, and Video Technology*, 21(3):251–263, July 1999.

[TCGK02] L. Thiele, S. Chakraborty, M. Gries, and S. Künzli. Design space exploration of network processor architectures. In *Network Processor Design 2002: Design Principles and Practices.* Morgan Kaufmann Publishers, 2002.

[TLP05] B.L. Titzer, D.K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 477–482, 2005.

[TPB98] T. Truman, T. Pering, and R. Brodersen. The infopad multimedia terminal: A portable device for wireless information access. *IEEE Trans. Comput.*, 47(10):1073–1087, 1998.

[TPS+05] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, Ph. Buonadonna, D. Gay, and W. Hong. A macroscope in the redwoods. In *Proc. 3rd International Conference on Embedded Networked Sensor*

Systems (SenSys), pages 51–63, New York, NY, USA, 2005. ACM Press.

[Tri01] Triscend Corp. Triscend E5 configurable system-on-chip, 2001.

[VGSR05] E. Vergetis, R. Guerin, S. Sarkar, and J. Rank. Can bluetooth succeed as a large-scale ad hoc networking technology? *Selected Areas in Communications, IEEE Journal on*, 23(3):644–656, 2005.

[WASW05] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: A wireless sensor network testbed. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN'05), Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS)*. IEEE, Piscataway, NJ, apr 2005.

[Wei91] M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, September 1991.

[WK02] G. Wigley and D. Kearney. Research issues in operating systems for reconfigurable computing. In *Proceedings of the International Conference on Engineering of Reconfigurable System and Algorithms(ERSA)*, pages 10–16. CSREA Press, Juni 2002.

[WLBW01] R. Wattenhofer, L. Li, P. Bahl, and Y.-M. Wang. Distributed topology control for power efficient operation in multihop wireless ad hoc networks. In *Proc. of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2001.

[WMF02] E. Welsh, P. Murphy, and J.P. Frantz. Improving connection times for Bluetooth devices in mobile environments. In *Proc. Int'l Conf. Fundamentals of Electronics Communications and Computer Sciences (ICFS 2002)*, March 2002.

[Wol02] W. Wolf. *Computers as Components: Principles of Embedded Computing System Design.* Morgan Kaufman Publishers, 2002.

[WP03a] H. Walder and M. Platzner. Online scheduling for block-partitioned reconfigurable devices. In *Proceedings of the International Conference on Design, Automation and Test in Europe (DATE)*, pages 290–295. IEEE Computer Society, March 2003.

[WP03b] H. Walder and M. Platzner. Reconfigurable hardware operating systems: From design concepts to realizations. In *Proceedings of the 3rd International Conference on Engineering of Reconfigurable Systems and Architectures (ERSA)*, pages 284–287. CSREA Press, June 2003.

[WTT+06] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using rpc for interactive development and debugging of wireless embedded networks. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, pages 416–423, New York, NY, USA, 2006. ACM Press.

[WZ04] R. Wattenhofer and A. Zollinger. XTC: a practical topology control algorithm for ad-hoc networks. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, pages 216–222, 2004.

[Xil02] Xilinx Inc. Virtex-II Pro platform FPGA handbook, 2002.

[ZBG98] X. Zeng, R. Bagrodia, and M. Gerla. Glomosim: a library for parallel simulation of large-scale wireless networks. In *Proc. Twelfth Workshop on Parallel and Distributed Simulation (PADS)*, pages 154–61, 1998.

[ZLT01] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength Pareto evolutionary algorithm for multiobjective optimization. In *Proc. EUROGEN 2001 – Evolutionary Methods for Design, Optimisation and Control with Applications to Industrial Problems*, 2001.

[ZSKD03] C. Zissulescu, T. Stefanov, B. Kienhuis, and Ed Deprettere. Laura: Leiden Architecture Research and Exploration Tool. In *Proceedings of the 13th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 911–920, 2003.

Acronyms

(AD)PCM	(Adaptive Differential) Pulse Code Modulation
ASIC	Application-Specific Integrated Circuit
CLB	Complex Logic Block
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
(C)SoC	(Configurable) System on a Chip
DAG	Directed Acyclic Graph
DSP	Digital Signal Processor
DVS	Dynamic Voltage Scaling
FPGA	Field-Programmable Gate-Array
KPN	Kahn Process Network
LUT	Look-up Table
PDA	Personal Digital Assistant
RISC	Reduced Instruction Set Computer
RSSI	Received Signal Strength Indicator
SDF	Synchronous Dataflow
UART	Universal Asynchronous Receiver and Transmitter
(V)HDL	Hardware Description Language
WSN	Wireless Sensor Network

Curriculum Vitae

Name	Matthias Dyer
Date of Birth	16 October 1976
Citizen of	Winterthur (ZH)
Nationality	Swiss

Education:

2002–2007	ETH Zurich, Computer Engineering and Networks Laboratory Doctor Thesis under the Supervision of Prof. Dr. L. Thiele
1997–2002	ETH Zurich, Department of Electrical Engineering and Information Technology Studies in Electrical Engineering and Information Technology Graduation as Dipl. El.-Ing. ETH
1992–1997	Mathematics and Science Gymnasium, Kantonsschule im Lee, Winterthur Graduation with Matura Type C
1990–1992	Secondary school in Winterthur, Switzerland

Professional Experience:

2002–2007	Research & Teaching Assistant at ETH Zurich
2000	Embedded Software Engineer at Intertec Contracting, Aarhus, Denmark
1998	IT Support at Boxler Informatic, Kloten, Switzerland
1997	Engineering Trainee at Sulzer, Winterthur
1997	Volunteer at Baumberger Electronic, Zurich