Reconfigurable Medium Access Control Protocols for Wireless Networks

Von der Fakultät für Elektrotechnik und Informationstechnik der Rheinisch-Westfälischen Technischen Hochschule Aachen zur Erlangung des akademischen Grades eines Doktors der Ingenieurwissenschaften genehmigte Dissertation

vorgelegt von

Xi Zhang, M.Eng
aus Shanxi, P.R. China

Berichter: Univ.-Prof. Dr. Petri Mähönen
Univ.-Prof. Dr.-Ing. Gerd Ascheid

Tag der mündlichen Prüfung: 04 Juli 2014

Diese Dissertation ist auf den Internetseiten der Hochschulbibliothek online verfügbar.
Abstract

Wireless networks are becoming ubiquitous. They have numerous applications also outside the cellular systems, for example in agriculture, multi-user games, health monitoring using body area networks, and security surveillance. Wireless networks operate using a limited and unstable radio resource, yet they must provide a reliable service for an array of varying applications and mobility patterns. These challenges are compounded by having to rapidly develop hardware platforms and software stacks as new applications emerge.

Medium Access Control (MAC) protocols facilitate access to the shared spectrum by defining rules by which wireless terminals communicate with each other. A runtime reconfigurable MAC protocol is able to change these rules dynamically. Supporting varying application needs, Quality of Service (QoS), available spectrum utilization in accordance to regulatory policies, and fair resource sharing among networks make runtime reconfigurable MAC protocols desirable. Traditionally, MAC protocols are optimized for one or very few specific operating scenarios, and are thus unable to guarantee satisfactory performance in increasingly complex and dynamic wireless environment. The goal of this work is to provide MAC protocols with a high level of runtime reconfigurability for any changing applications and spectral environment.

In this dissertation, we present a MAC protocol development framework and an associated toolchain for on-the-fly MAC protocol realization and reconfiguration. MAC functionalities are decoupled into a set of common components which are used to construct MAC protocols. To validate the applicability of our approach, we implemented and evaluated our framework and toolchain in two significantly different application areas: cognitive radio networks and wireless sensor networks. The component-based architecture enabled realization of both classical and new MAC protocols with very low overhead. Runtime MAC protocol reconfiguration is enabled in two ways: small-scale reconfigurations using MAC parameter tuning, and macroscopic reconfigurations through component-based MAC protocol reconstruction. We evaluated our implementations using hardware platforms in realistic environments. The experimental results on MAC protocol reconfiguration show that the developed framework improved throughput and packet delivery ratio up to 400% in an unstable spectral environment. Adapting to varying application requirements is achieved, and seamless QoS is offered under highly varying conditions. In addition, our toolchain enables parallel execution of independent MAC components on many-core architectures. We found parallel execution to improve MAC performance especially for computationally intensive algorithms.
I would like to express my gratitude to Prof. Dr. Petri Mähönen, who has provided me with valuable guidance in the course of my pursuit to the doctor title. He is always supportive to my ideas and has given me the freedom to explore the unknowns. I am very happy that I can finally answer him the question he has been asking during the past six years: Yes, it is ready. I would also like to thank Prof. Dr. -Ing. Gerd Ascheid for his interest in co-examining this work. His feedback will certainly be beneficial.

I would like to especially thank Dr. Junaid Ansari for a long term collaboration on various research topics. His willingness to share new ideas leads to fruitful discussions and sometimes exciting research results. I have also learnt a great deal of both technical and soft skills from him. He has certainly helped in shaping this thesis the way it is now. I would like to thank my students, Guangwei Yang, Obaid Salikeen, Xuan Zhu, Luis Miguel Amoros Martínez, Noemi Arbós Linio and Manish Arya, who had the courage to work with me. Their contributions are important for this dissertation. I would like to thank my colleagues at iNets who provided a friendly and stimulating working environment. They are always willing to go out of their way to help. I would also like to thank Gero Schmidt-Kärst for his technical support with all kinds of experimental equipments. All the experiments and demonstrations I have done would have been more painstaking without his support.

I am thankful to RWTH Aachen University, European Union (WASP-IST-034963 and 2PARMA-FP7-ICT-2009-4-248716 projects) and German Research Foundation’s UMIC research centre for providing necessary funding grants.

I would also like to thank my friend Zhongjie for proofreading part of this dissertation and providing valuable comments. I will certainly return the favour when he finishes his dissertation. I thank my parents for their support during my PhD research. They have always been confident in me, even during the times when I was in doubt. I am very glad that now I am one step closer to be a ”da bo shi” and to make them proud. Last but not least, I would like to thank Antti, for always being critical about my papers, my codes, and my talks. At the same time, he has been extremely supportive. In his own way, he helped me deliver better results and finish this dissertation on time. And I will not start listing down the nice dishes he has cooked in the past five years which have definitely contributed to this work, as the list might be longer than the dissertation.
**CONTENTS**

**ABSTRACT**

I

**ACKNOWLEDGEMENTS**

III

**CONTENTS**

V

**1 INTRODUCTION**

1.1 MOTIVATION ................................................. 1
1.2 MAJOR CONTRIBUTIONS ....................................... 3
1.3 DISSERTATION STRUCTURE ..................................... 5

**2 RELATED WORK**

2.1 FLEXIBILITY AND ADAPTATION IN WIRELESS NETWORKS ....... 7
2.2 FLEXIBLE MAC DEVELOPMENT ON COMMODITY IEEE 802.11 HARDWARE ............................................. 8
2.3 ADAPTABLE MAC PROTOCOLS FOR WIRELESS SENSOR NETWORKS .................................................. 10
2.4 MAC SCHEMES FOR SOFTWARE DEFINED RADIOS ............. 13
2.5 COMPONENT-BASED DESIGN APPROACH FOR FLEXIBLE PROTOCOLS ............................................. 17

**3 DECOMPOSABLE MAC FRAMEWORK**

3.1 DESIGN GOALS ................................................ 22
3.2 LIBRARY OF REUSABLE COMPONENTS ......................... 24
3.2.1 TIMERS .................................................. 24
3.2.2 CHECKSUMS ............................................... 25
3.2.3 CARRIER SENSING ......................................... 26
3.2.4 RADIO CONTROL ........................................... 27
3.2.5 FREQUENCY SELECTION ..................................... 27
3.2.6 TRANSMISSION POWER SELECTION ....................... 28
3.2.7 RANDOM NUMBER GENERATOR ............................. 28
3.2.8 SEND FRAME ............................................... 28
3.2.9 RECEIVE FRAME ............................................ 29
3.2.10 Secondary Level Components ................................... 29
3.2.11 Block Diagrams for Sample MAC Protocols ............ 31
3.3 Framework Implementation and Evaluation Results .... 33
  3.3.1 Decomposable MAC Framework Implementation
      for WARP Boards ........................................... 33
  3.3.2 Sample MAC Protocol Realizations on WARP Board
      Using Decomposable MAC Framework ...................... 36
  3.3.3 Decomposable MAC Framework Implementation
      for Sensor Nodes ........................................... 46
3.4 MAC Protocol Designer .............................................. 52
  3.4.1 User Interface ............................................. 53
  3.4.2 XML Modelling ............................................. 54
  3.4.3 Code Loader ................................................. 55
  3.4.4 TinyOS 2.x Code Generation .............................. 55
  3.4.5 Performance Evaluation .................................. 56
3.5 Summary and Discussions .......................................... 57
4 A Toolchain for Runtime MAC Protocol Realization ...... 59
  4.1 TRUMP System Design .......................................... 60
    4.1.1 Component-Based Framework ............................ 60
    4.1.2 Wiring Engine ........................................... 61
    4.1.3 Meta-Language Descriptor .............................. 69
    4.1.4 Meta-Compiler .......................................... 70
  4.2 Evaluation Results of TRUMP on WARP Boards .......... 71
    4.2.1 Memory Usage ........................................... 72
    4.2.2 Execution Time Overhead .............................. 72
  4.3 Reconfigurable MAC Protocols for Resource Constrained
      Platforms .................................................... 75
    4.3.1 MAC Component Library ................................ 77
    4.3.2 Execution Scheduler ................................. 77
    4.3.3 Meta-Language Descriptions and the Parser .................. 79
    4.3.4 Command Shell ........................................ 79
  4.4 Evaluation Results on Sensor Nodes ....................... 80
    4.4.1 Memory Consumption .................................. 81
    4.4.2 Execution Time ........................................ 82
    4.4.3 Reconfiguration Cost .................................. 83
  4.5 Summary and Discussions ..................................... 84
5 MAC Protocol Optimization ........................................ 87
BIBLIOGRAPHY 151
1

INTRODUCTION

Wireless Medium Access Control (MAC) protocols define the rules for the wireless terminals to access the medium in an orderly manner. MAC protocols govern the communication among terminals and play a crucial role in ensuring fair and efficient sharing of wireless spectrum [1]. This dissertation focuses on introducing reconfigurability to MAC protocols for different types of wireless networks. Reconfigurability refers to the ability to modify the medium access rules and procedures. Reconfiguration can be performed by both the MAC protocol users and the protocol itself. MAC protocols should be flexible, meaning that MAC protocols should allow tuning of behaviour, in order to realize reconfiguration. MAC adaptability is enabled if self-reconfiguration is carried out. We present a component-based MAC framework and an associated toolchain which together realize runtime flexible reconfiguration of MAC protocols. We show that our tools enable desired MAC features such as runtime performance optimization and parallelization of MAC processes for multi-core platforms. In this dissertation, terms including MAC protocol, MAC scheme, and MAC algorithm are used interchangeably.

1.1 MOTIVATION

There has been a surge of wireless technologies over the last decade, which offers higher data transmission rates and enables various mobile applications. This phenomenon leads to an increasing popularity of a wide range of applications and rapid development of both established and new types of networks. As a result, hardware platforms for wireless terminals, wireless network standards, and spectrum regulatory policies are constantly evolving [2]. MAC protocols play a vital role in delivering satisfying network performance for a wide range of applications on different types of networks and hardware platforms [3]. In this dissertation, we identify three key challenges for today’s MAC protocol design and implementation.

• Over the past years, due to the diverse needs of a wide range of wireless applications and deployments, numerous MAC protocols have been designed for various types of wireless networks including different kinds of ad hoc networks, wireless sensor networks, body area networks, and cognitive radio networks [4–9]. Since MAC protocols require close inter-
action with radio frontend and physical (PHY) layer, MAC protocol prototyping on real hardware platforms requires the MAC designer to know about platform specific hardware characteristics and programming environment. Therefore, over-the-air testing of MAC protocols on real testbeds is a cumbersome task. Most of the MAC protocol proposals are based on theoretical analysis and computer-based simulation [3]. Theoretical models often use over-simplified assumptions to ensure mathematical tractability while wireless network simulators are not able to provide accurate physical layer information and realistic channel conditions [10]. How to enable fast and easy MAC protocol prototyping on target hardware platforms is a challenge in the MAC protocol development community.

• As the number of wireless networks increases, the complexity in spectrum sharing among networks is increasing. Although networks such as Global System for Mobile communications (GSM) and TV broadcast are licensed with specific spectra, numerous small scale and/or newly emerged networks do not have their dedicated frequency bands. They are required to either survive in Industrial, Scientific, and Medical (ISM) bands, which are available for all users, with many other networks; or act as white space devices to opportunistically use vacant slots in the already licensed bands [11,12]. Under these circumstances, wireless terminals and networks would experience high fluctuation in bandwidth availability. Most of the classically designed MAC protocols are assumed to operate in, and thus are optimized for, specific network conditions. These MAC protocols are static, which do not allow reconfiguration and modification, since changing MAC behaviour would result in suboptimal performance for the specific network condition. However, a static MAC scheme is not sufficient to deliver desired performance in a dynamic spectrum environment [13,14]. Furthermore, application requirements and network traffic patterns for a wireless network can vary both temporally and spatially. A static MAC scheme also fails to cope with the changing demands from applications. Therefore, how to handle the ever changing operating environment and demanded Quality of Service (QoS) at the MAC layer is another challenge.

• In order to identify the precise spectral condition and select the most suitable resource for communication in a highly unstable spectral environment, sophisticated and computationally intensive sensing mechanisms are needed. In the area of Cognitive Radio Networks (CRNs), cognitive radio terminals are generally required to process data over a wide-band to identify spectrum opportunities. To improve utilization of spectrum, fast data processing capability is essential [15]. Long Term Evolution (LTE) [16], which is the latest standard in mobile network technology,
1.2 MAJOR CONTRIBUTIONS

uses dynamic resource scheduling algorithms to improve the utilization of limited radio spectrum resources. The complexity of resource scheduling algorithms, stringent real-time response and rate-guarantee requirements lead to a high computational demand at MAC layer [17]. Multicore processor architecture for wireless terminals is a mean to provide the computational capability for fast data processing and meeting timing constraints for real-time systems [17, 18]. Therefore, how can MAC protocols benefit from a multi-core architecture efficiently is the third challenge.

1.2 MAJOR CONTRIBUTIONS

Facing the challenges as described above, we have designed and implemented a framework with a toolchain for easy MAC protocol prototyping for a wide range of wireless networks. The toolchain enables efficient runtime reconfiguration and optimization of MAC protocols. It is also capable of speeding up MAC execution by exploiting parallelism within MAC functionalities on multi-core/many-core platforms. The major contributions of this dissertation are listed as below.

• Enabling Easy MAC Protocol Prototyping

We have introduced a component-based framework (Decomposable MAC Framework) for rapid MAC protocol realizations. Common MAC functionalities are identified from an extensive study over a wide range of MAC protocols for different kinds of wireless networks. These common functionalities are treated as building blocks so that a particular MAC scheme can be realized by simply binding the components together. This approach reduces the design and debugging effort for MAC protocol prototyping since all the provided MAC components are well tested as part of the framework. The MAC designers are exempted from learning platform specific programming environment and hardware details. Based on Decomposable MAC Framework, we have also developed a MAC protocol designing tool (MAC-PD). MAC-PD allows users to design and prototype MAC protocols in the form of flowcharts. The code for a selected target platform is automatically generated based on the flowcharts. The MAC protocols implemented using Decomposable MAC Framework show over 80% code reuse and similar performance characteristics as compared to their monolithic hand-coded counter-parts.

• Runtime MAC Protocol Reconfiguration and Optimization

Decomposable MAC Framework enables MAC protocol realizations in the form of a set of MAC building blocks bound together with a specific execution flow. Changing from one MAC behaviour to another is
realized by selecting a different set of components and/or modifying
the connection logic among them. This Lego like concept facilitates run-
time reconfiguration and optimization of MAC protocols. Runtime opti-
mization is important for protocols working in a dynamic environment
with unpredictable changes in terms of spectrum availability, network
structure, application requirements, etc. We have introduced a toolchain
for runtime protocol realization (TRUMP) which enables runtime opti-
mization. This toolchain includes a simple MAC meta-language for
MAC protocol design which specifies the MAC components used and
the connection logics, a meta-compiler which translates the MAC proto-
kol description to executables on the target platforms, and a Wiring En-
gine which governs the interconnection among MAC components and
MAC protocol executions. Using TRUMP, MAC protocols can be re-
configured with great flexibility at runtime. We have also extended our
meta-compiler capability to further automate protocol optimization pro-
cedure. The user can specify MAC performance optimization options
such as power consumption and data reliability through the compiler.
The toolchain selects a suitable set of MAC components and param-
ters at runtime according to the user specification and operating envi-
ronment. Furthermore, a co-operative MAC scheme is developed as a
plug-in of the toolchain. The co-operative MAC module ensures that the
optimization at a node is reflected appropriately throughout the network
so that a network-wide optimization could be achieved. Our experimen-
tal results show that our MAC protocols are able to offer relatively stable
performance under highly interfered spectral conditions.

• Speed Up MAC Execution on Multi-core/Many-core Platforms

Multi-core/many-core platforms can be used for wireless terminals to
achieve the timing requirements of new wireless standards, and to pro-
vide the computational power demanded by heavy wide-band data pro-
cessing, optimization schemes, and resource scheduling algorithms. Our
toolchain TRUMP provides the capability to speed up MAC executions
by exploiting multi-core hardware architecture and parallelized MAC
processes. Interdependencies among MAC processes are identified and
stored. TRUMP schedules MAC processes for sequential or parallel exe-
cution based on the dependencies and the hardware resource availabil-
ity. We have also looked into parallelization of a number of machine
learning based computationally intensive algorithms used for MAC pro-
tocol optimization and resource allocation. Our results show that we are
able to achieve an up to 85% improvement on a many-core platform in
convergence time for a genetic algorithm used for MAC/PHY param-
eter optimization. We have also shown when using swarm intelligence
algorithm for channel allocation, it is 2-6 times more likely for a node
with more computing power to get a desirable channel than a node with limited computing power.

- **Realistic Implementation and Evaluation**

  All the framework, tools and protocols discussed in this dissertation have been realized on either commercially available hardware platforms or a cycle accurate hardware emulator. All the experiments are conducted in a realistic wireless environment in an office. Our approach to realize re-configurable MAC protocols focus on a practical aspect where we have achieved a high percent of code re-usage and easy portability among platforms.

### 1.3 Dissertation Structure

The dissertation is structured as follows. We describe related work on flexible and reconfigurable MAC protocols for different types of wireless networks and hardware platforms in Chapter 2. In this chapter, we present research work on flexible MAC protocol development for Wireless Local Area Networks (WLANs) on IEEE 802.11 NIC, Wireless Sensor Networks (WSNs) and Software Defined Radio (SDR) platforms. A discussion on component-based design approach to achieve flexible network protocols is also included. In Chapter 3, we introduce Decomposable MAC Framework. The set of identified common MAC components is described in detail. Sample MAC realizations using Decomposable MAC Framework on both Wireless Open-Access Research Platform (WARP) boards and sensor nodes are presented. We also describe MAC-PD for rapid prototyping of MAC algorithms. TRUMP, a toolchain for runtime MAC protocol realization, is introduced in Chapter 4. The design and implementation of TRUMP for both WARP SDR boards and sensor nodes are presented. While the design philosophy for both types of platforms is the same, sensor nodes impose a stricter constrain on memory and computation resource. Both implementations are evaluated in terms of execution speed and memory overhead, and protocol reconfiguration efficiency. Chapter 5 describes a collection of work done to enable and improve runtime optimization capability of MAC protocols using TRUMP. A compiler assisted approach is presented which supports MAC protocol optimization with runtime monitoring of application requirements. We also present a co-operative scheme which facilitates network-wide optimization. The results on using multi-core/many-core platform for realization of complex MAC procedures are discussed in Chapter 6. We discuss the hardware platform architecture in general for the new generation of wireless communications and identify the trend of the architectural design. Genetic algorithms and swarm intelligence algorithms which have been proposed for MAC protocols for parameter optimization and resource allocation are implemented on a many-core computing
fabric cycle accurate emulator. We conclude the dissertation and discuss possible future work directions in Chapter 7.
In this chapter, we present the state-of-the-art efforts made for wireless networks to support a wide variety of distributed applications with increasing complexity of network conditions, varying mobility patterns and limited spectrum resource. After briefly introducing smart adaptation efforts enabled by applications and middlewares, we focus on the work done in achieving flexible MAC schemes for wireless networks. Particularly, we present the work in three areas: WiFi network which is restricted to IEEE 802.11 standard and its compliant commodity hardware; CRN and SDR platforms which belong to a relatively new research area; sensor network which is commonly associated with resource constrained platforms. Although these three types of networks face different challenges from their own perspectives, flexibility and adaptability are commonly required due to the nature of dynamicity in wireless networks.

2.1 Flexibility and Adaptation in Wireless Networks

In 1990s, wireless networks were gaining popularity due to the easy installation procedure and the mobility they offer. As we shift from wired networks to wireless networks, we face new challenges in both application and networking protocol stack designing. Wireless networks are inheritably unstable in providing QoS as compared to wired networks due to varying channel conditions. Traditionally for wired networks, applications impose strict requirements on QoS to the network. For example, low latency is a strict requirement for real-time applications while data reliability is demanded for secure message exchanges. These requirements are difficult to be fulfilled by wireless networks. Therefore, additional considerations and mechanisms are to be introduced to satisfy QoS requirements at all time.

Several research efforts have incorporated flexibility within the application by setting a loose bound for QoS and adaptively re-adjusting the QoS within the bound in order to accommodate network dynamics and user mobility [19]. Naghshineh et al. have presented an adaptive framework for multimedia applications which bridges the application needs and the networking requirements [20]. It breaks down multimedia connection into multiple substreams with individual QoS requirements, and schedules the substreams to the net-
work adaptively with a mapping of the QoS requirements to different levels of network resource availability.

In order to cope with dynamic environment and application needs, adaptation has been incorporated into middleware as well. Adaptive middleware employs four key software technologies to support adaptation [21]: 1) computational reflection which enables middleware to inspect, reason and self-adapt at runtime [22]; 2) component-based design which decomposes middleware functionalities and enables easy and dynamic structural modification [23]; 3) aspect-oriented programming which decouples cross-cutting concerns such as QoS, energy consumption and fault tolerance [24]; and 4) software design patterns which allow reuse of adaptive design patterns [25]. These software adaptation technologies have greatly influenced our approach to enable reconfigurable MAC protocol realizations which we are describing in this dissertation. Adaptive middleware projects are usually designed focusing on one or several aspects. A Common Object Request Broker Architecture (CORBA) [26] based adaptive middleware platform is proposed by Blair et al. [27] to support mobile multimedia applications. The authors suggested that adaptive middleware platforms are essential in order to accommodate the demanding requirements for QoS adaptation which are imposed by these types of applications. Sun et al. have presented the design and implementation of a connectivity management middleware which aims at providing stable end-to-end connection in a wireless mobile environment [28].

While the adaptation at application layer is necessary and adaptive middleware solutions have shown their effectiveness in supporting the QoS requirement for wireless networks, the full potential of adaptation can be further explored. The data link layer, which is responsible for sharing network resources and plays an important role in deciding the level of services being provided to the application, should be incorporated in the adaptive solutions. Bianchi et al. have discussed on a programmable MAC framework to achieve adaptive QoS support [29]. In this thesis, we share the same view that by giving flexibility to the MAC layer, we are able to provide adaptable solutions to various types of wireless networks and enhance the MAC layer performance. We provide an enabler for adaptive MAC protocol realizations. In our work, we have assumed that the adaptation we have taken at the MAC layer is self-contained and acts independent from the application, i.e. the application imposes only QoS requirements without directly determining the adaptation procedures taken by the MAC layer.

2.2 FLEXIBLE MAC DEVELOPMENT ON COMMODITY IEEE 802.11 HARDWARE

MAC algorithms are classically designed and implemented in a monolithic fashion with tight coupling to the underlying hardware platform, which restricts
the reconfigurability and adaptation aspects required in wireless networks. For WLAN which is a very popular type of ad hoc networks, IEEE 802.11 standards have been strictly reinforced on both the PHY and MAC layer to ensure compatibility of devices from different manufacturers and fair spectrum access from numerous resource contenders. In order to meet the strict timing constraints on data processing and operations imposed by the standards, all the commercially available IEEE 802.11 Network Interface Controllers (NICs) are rigidly optimized and offers very limited space for reconfiguration.

SoftMAC [30] is a system which offers flexible MAC layer with pre-defined PHY layer. It is built on-top of a commodity IEEE 802.11a/b/g Atheros networking card and uses open-source MadWifi [31] drivers. SoftMAC exposes the flexibility offered by the Atheros chipset and provides a driver which enables control over the MAC layer. For example, the standard 802.11 MAC features such as Request-To-Send (RTS) Clear-To-Send (CTS) handshake, auto acknowledgement (ACK) and retransmissions, virtual carrier sensing, etc. can be disabled so that the default MAC layer performs basic packet sending and receiving functionality. An additional MAC layer can be implemented on top to exercise schemes such as Time Division Multiple Access (TDMA) based MAC protocols. SoftMAC decouples the MAC behaviour from the commodity wireless adapters. It offers researchers freedom to experiment with different types of MAC protocols over the IEEE 802.11 NIC. MultiMAC [32] extends SoftMAC system in terms of adding the idea of runtime adaptability. Multi-MAC allows multiple MAC layers to coexist concurrently in the network stack. These pre-defined standalone protocols can be switched at runtime on a per packet basis. The MultiMAC framework selects a MAC protocol in its MAC pool best suited for the particular network conditions. Although this scheme provides flexibility and adaptability features, the design choice is confined to a subset of pre-selected protocols and it can at best approximate the closest fit. Furthermore, many communication systems and devices are limited in terms of onboard memory to support a wide range of predefined MAC schemes. Also motivated by SoftMAC, MadMAC [33] is a kernel-mode driver to build new MAC protocols on commodity 802.11 NICs. A TDMA based MAC protocol is built with reconfigurable slot structure and packet format. The frame timings at the transmit and receive interfaces are controlled through the Mad-Wifi driver for Atheros based NICs in order to improve the overall throughput. However, the scope of configurations achieved by the parameter tuning based approach is limited to the permitted range of parameters. A similar work Flex-MAC [34] has been demonstrated on commodity hardware which shows that a software based IEEE 802.11b implementation performs closely to a commercially available hardware based implementation with only 3% bandwidth loss in Transport Control Protocol (TCP) due to imperfect timing of scheduling [35].

As the number of wireless devices operating in the limited ISM band grows exponentially, the wireless network condition becomes more unpredictable. The traditional approaches to provide desired QoS with varying level of band-
width availability by rate adaptation or changing basic medium access procedure on a fixed channel are no longer sufficient. MAC protocols are shifting to using multiple channels to increase the possibility of providing the required services. Sharma et al. designed FreeMAC [35] – a multichannel MAC development framework exposing the MAC interfaces for the protocol developer on top of the standard IEEE 802.11 compliant hardware platforms. This work aims at supporting frequent channel switching and allows better timing characteristics for spectrum agile MAC schemes. A simple multi-channel TDMA MAC protocol is implemented as a proof of concept, although synchronization among nodes is done through Ethernet interface which is not practical in real life network deployment. Furthermore, the MAC implementation is dependent on the functions provided by the underlying hardware such as the Atheros hardware timers which makes the MAC implementation difficult to be realized on other platforms. Although the timing characteristics of TDMA based MAC implementation in software over commodity hardware have been improved in later research work [37], these MAC protocols are usually designed for specific applications and lacks the flexibility required by a highly dynamic spectrum environment. These efforts on development of overlay software modules over commodity hardware are overall constrained by the programming interfaces provided by the underlying drivers.

Tinnirello et al. introduced the idea of wireless MAC processors [38] on commodity WLAN cards. MAC protocols are implemented in the form of an extended finite state machine. It supports runtime injection of MAC state machine to program MAC operation without interrupting the MAC service. It has distinguished itself from the previous overlay software modules by providing the easy runtime programmability of MAC layer on commodity hardware. The underlying MAC architecture is based on our Decomposable MAC Framework as described in Chapter 3 of this dissertation. As an extension work, Bianchi et al. introduced MAClet, which is a coded state machine with an initial state description to be fed on the wireless device [39]. An overlay software control framework is developed over the wireless MAC processor to move and launch MAClets to enable dynamic MAC stack reconfiguration. Although the MAC processor approach offers great flexibility in programming the WLAN cards, the target scope of MAC protocols is currently limited to IEEE 802.11 alike ones.

2.3 ADAPTABLE MAC PROTOCOLS FOR WIRELESS SENSOR NETWORKS

Flexibility and adaptability at MAC layer are equally important for WSNs as for ad hoc networks. In addition to the dynamic spectrum environment often experienced by WSNs since they commonly operate in ISM bands, MAC protocols for WSNs are required to be energy efficient in general due to the
limited battery life-time of sensor nodes. Numerous adaptive MAC protocols have been proposed focusing on addressing the energy efficiency issue. One approach we have taken is from the perspective of hardware capability. Benefiting from the characteristics of radio chips operating in low and high frequency bands, we have designed a dual-radio platform which supports high data transmission rate using one radio chip and low-power channel sniffing using the other radio chip [40]. A multi-radio MAC protocol is implemented accordingly which adaptively choose the radio chip to carry out different tasks based on the traffic load and latency requirement. Many more approaches have been explored in terms of protocol software designs.

MAC parameter tuning is the main method in achieving adaptivity. MAC protocols for WSNs often exercises Low Power Listening (LPL) where nodes wake up periodically to listen to a channel for a short period of time. Duty cycle refers to the ratio of the active listening time to the period duration, i.e. sum of active duration and sleep interval. T-MAC [41] is an IEEE 802.11 inspired MAC protocol with adaptive duty cycle. It automatically adapts the duty cycle to the current network traffic. Instead of using a fixed-length active period, T-MAC uses a time-out mechanism to dynamically determine the length of the active period. If a node does not detect any activity within the time-out interval, it goes to sleep. This scheme reduces the amount of energy wasted in idle listening in which the receiving nodes wait for potential incoming packets. nanoMAC [42] provides a means to select one of the three supported fixed duty cycles based on the application traffic requirements in order to save the idle mode power consumption.

WiseMAC [43, 44] is a preamble-sampling based MAC protocol which is adaptive to traffic load. In preamble-sampling MAC protocols, a wake up preamble is transmitted before every packet to ensure that the receiver which is performing LPL, will be awake when the data portion of the message arrives. To minimize the wake up preamble overhead, sensor nodes learn the offset between the sampling schedule of their direct neighbours and their own in WiseMAC. The preamble length varies with the time elapsed since the last acknowledgement message was received from the destination. WiseMAC is adaptive to traffic load since the length is dependent on the time interval between packets. When the traffic load is high, means a node transmits to another node frequently, the length of the preamble is hugely reduced thus giving high energy efficiency in this situation. When the traffic load is low, the length of the preamble is limited by the sleep interval of the receiver and thereby providing ultra-low power consumption. Inspired by WiseMAC, MFP-MAC [45] and X-MAC [46], we have designed a Traffic aware MAC (TrawMAC) protocol [47], which is a multi-mode MAC scheme and adapts its behaviour in packet transmission to the changing traffic and network conditions. TrawMAC optimizes energy consumption by exploiting the shared traffic information across routing and MAC protocol layers with minimum compromises in latency and packet delivery ratio.
MaxMAC [48] offers a smart scheme to adaptively change the wake-up rates in reaction to the offered traffic load. MaxMAC keeps track of the incoming traffic at a node. If the traffic rate increases over a pre-defined threshold, extra wake-ups are introduced. The simulation results show higher throughput and lower latency as compared to T-MAC and WiseMAC. The dynamic duty cycling of MaxMAC has an edge over static duty cycling schemes such as in A-MAC [49,50] where a node doubles the active periods when higher traffic volumes are experienced. Bac et al. [51] propose a tree based scheme with adjustments of duty cycles based on the traffic loads in the network. The nodes are synchronized and use a super-frame structure at each level with distinct topology. This scheme has its limitations in the presence of high network dynamics and is therefore not suitable to many low-power embedded network deployments. StrawMAC [52] is a contention based protocol and exercises RTS-CTS backoff strategy. It is designed to operate in low duty cycles, however in case of sporadic traffic surges, it tries to adapt its duty cycle by estimating the length of the data packet through control frames.

Besides the above mentioned duty cycle based adaptation approach, other parameters and mechanisms have been investigated in WSN research to allow better flexibility and adaptability of MAC protocols. Receiver-Based Auto-Rate Protocol (RBAR) [53] allows modification of modulation schemes at runtime. It estimates channel quality, executes runtime adaptation mechanism in the receiver side and notifies transmitter to choose appropriate modulation schemes during RTS-CTS exchange. This approach achieving limited MAC performance improvement because it only allows parameter adaptation without the capability to change the functional behaviour of MAC protocols. Hybrid MAC Protocols use different MAC protocols together as one MAC protocol. Z-MAC [54] is a hybrid protocol which combines the strengths of TDMA and Carrier Sense Multiple Access (CSMA) while offsetting their weaknesses. Each node is statically assigned a time slot by Z-MAC just like TDMA based MAC protocols, but unlike TDMA, a node can transmit in both its own time slot and other slots assigned to other nodes. The contention resolution scheme used by Z-MAC has an effect of switching between CSMA and TDMA depending on the level of contention, thus achieving the performance of CSMA under low contention and that of TDMA under high contention. Z-MAC is shown to out-perform non-adaptive B-MAC [55] in high traffic load scenarios. However, Z-MAC has a fairly complex signalling mechanism and in low traffic conditions, its control overhead starts to dominate. Funneling-MAC [56] also uses the hybrid CSMA/TDMA principle to effectively handle the traffic load near sink node(s). It has a complex signalling mechanism and suffers from network dynamics. Rate Adaptive Hybrid MAC Protocol (RAH-MAC) [57] combines polling and contention based MAC protocols in order to benefit from the advantages of each one of them.

Related to the idea of combining protocols, Meta-MAC [14] is a systematic and automatic method to dynamically combine any set of existing MAC
protocols into a single upper layer. It achieves the performance of the best protocol without knowing in advance which of them will match the potentially changing and unpredictable network conditions. In addition, this optimization works without any centralized control or any exchange of messages, using only local network feedback information. Meta-MAC is introduced as a higher layer above existing MAC layer. It computes the decision of the best protocol to be used. However, this technique has a large memory footprint, large implementation effort and redundancy in source code. C-MAC \cite{58} is a highly configurable MAC protocols for WSN. It offers tunable parameters to application programmers to configure MAC protocols suiting to application needs. The parameter set includes synchronization, contention, error detection, acknowledgement, etc. C-MAC has taken a similar approach as our dissertation and WMP \cite{38} in decomposing MAC protocols into basic functionalities. It extracts state machines for protocol configuration. C-MAC is designed for IEEE 802.15.4 \cite{59} compliant radios. It focuses on giving the application programmers the ability of MAC protocol configuration instead of fully automated system level self-configuration and adaptation. Therefore, no runtime configuration capability is provided.

2.4 MAC SCHEMES FOR SOFTWARE DEFINED RADIOS

To enrich a wider scope of wireless MAC protocols with greater flexibility, various research activities have been carried out in the SDR community in both hardware platform design and software protocol stack development. SDR is defined by International Telecommunication Union (ITU) to be a radio transmitter and/or receiver employing a technology that allows the RF operating parameters including, but not limited to, frequency range, modulation type, or output power to be set or altered by software, excluding changes to operating parameters which occur during the normal pre-installed and predetermined operation of a radio according to a system specification or standard \cite{60}.

Numerous platforms have been designed and implemented for flexible PHY and MAC layer realization for highly dynamic and agile systems such as CRNs and active networks. The Universal Software Radio Peripheral (USRP) platforms \cite{61} are computer-hosted software radios. USRP translates analog radio signals to digital baseband complex samples which are then transferred to the host processor for further signal processing. USRP is often used with GNU Radio \cite{62} software framework and together provide means for flexible protocol implementations. Iris (Implementing Radio in Software) \cite{63} is an architecture for building highly reconfigurable radio networks in software. It operates on, but not limited to USRP \cite{64}. Despite the high level of flexibility and easy reconfigurability, software based implementation of protocols does not meet the performance characteristics demanded by the applications and the standards \cite{9,30}. The Sora platform \cite{65} exploits sophistica-
ted parallel programming techniques on multi-core Central Processing Unit (CPU) architectures to achieve throughput comparable to IEEE 802.11 Commercial Off-The-Shelf (COTS) hardware. However, any protocol stack modifications can be complicated to carry out due to sophisticated computation distribution on multi-core processor in an effort to meet real-time requirements. WARP [66] and OpenAirInterface [67] platforms use a hardware-software co-design philosophy where time critical operations are implemented in Field-programmable Gate Array (FPGA) to accelerate the process. Dutta et al. have proposed an architecture specifically for software defined cognitive radio [68] which concentrates on realization of an Orthogonal Frequency Division Multiplexing (OFDM) based radio PHY layer for easy adaptation. Unfortunately not much attention has been given to defining methodology for flexible MAC design. Since PHY and MAC layers interact closely, the benefits of an adaptive physical layer can be shown to a greater extent if the MAC layer can adapt accordingly and vice versa.

Cognitive Radios (CRs) are regarded as the technology means to recognize and utilize the available wireless spectrum. Mitola [69] anticipated that cognitive radios would become aware of the context and environment, and thus would be able to optimize their behaviour. Cognitive radio advocates spectrum sharing and coexistence with other networks. These unique characteristics have imposed new challenges on architecture and protocol design, especially for MAC layer. In the area of MAC protocol design for cognitive radio networks, there are both centralized and distributed MAC designs [7,8]. MAC protocols in infrastructure based networks require a central controller for managing network activities, including gathering, distribution and processing of data, synchronization among nodes, etc. IEEE 802.22 [70] standard for Wireless Regional Area Network (WRAN) is a good example in this category. Compared to distributed protocols, centralized MAC approaches typically demand simpler hardware and software capabilities for Secondary Users (SUs), i.e. CR nodes. Furthermore, MAC protocols relying on a centralized infrastructure impose stringent requirements in terms of sensing and coordination on the infrastructure. Complex mechanisms for spatiotemporal spectrum sensing and active coordination among nodes are needed. Numerous centralized protocols, such as [71–73], have been designed. Lien et al. have proposed a protocol based on the CSMA principle for underlay systems [74], where a SU activity is permitted with simultaneous Primary User (PU) activity as long as the interference caused to a PU is confined to a radio technology dependent threshold level. The protocol enables co-existence by adjusting the transmit power and data rates of CR nodes. For infrastructure based MAC protocols, the intelligence for adapting protocol parameters lies at the controlling centre. Zou et al. describe a game-theoretic approach for Dynamic Spectrum Access (DSA) driven MAC [75]. They propose a cluster based approach to centrally coordinate nodes through an out-of-band Common Control Channel (CCC). Chowdhury et al. suggest a CCC based scheme to coordinate CR nodes using OFDM for
2.4. MAC Schemes for Software Defined Radios

exchanging network information [76]. While performance results are provided in a custom simulation environment, implementation of the scheme on a real CR testbed has not yet been reported. CR-MAC [77] uses a multichannel scheme for providing access to SUs and is based on IEEE 802.11 Power Saving Model (PSM).

The other category of MAC protocols for CRNs is designed for the decentralized operation. The CR nodes gather spectrum information either locally or through cooperation with their neighbouring nodes. These protocols have different assumptions on the hardware platform and the network capabilities. Some of these schemes assume that cognitive users are equipped with multiple transceivers and nodes have the capability to access multiple channels simultaneously and select the best channel [78–80]. Single-Radio Adaptive Channel (SRAC) algorithm [81] enables dynamic channel access for legacy MAC protocols. SRAC proposed adaptive channelization, where a radio dynamically combines multiple fixed channels based on its needs to form a new channel. Simulation results using QualNet network simulator have been presented to show that SRAC is able to cope with spectrum jamming. IEEE 802.11 Distributed Coordination Function (DCF) inspired CREAM-MAC protocol [82] adopts a cooperative strategy and embeds channel information in control packets. It exercises a four-way handshake over the control channel to avoid multichannel hidden terminal problem. Salameh et al. [83] propose COMAC, a non-cooperative distributed protocol, which does not require coordination among PUs and CR nodes for opportunistic channel access. The protocol uses a probabilistic interference model among PUs and CR nodes in Rayleigh faded channels and aims at guaranteeing a certain statistical performance level for PUs at different traffic loads generated by CR nodes.

Some of the decentralized MAC protocols assume that all the entities in a network have an always available CCC for exchanging control information and establishing agreement on the selection of data transmission channel between the transmitter-receiver pairs. For example, inspired by IEEE 802.11 PSM and using a CCC, MMAC-CR [84] exercises an energy efficient distributed scheme for utilizing the licensed spectrum while protecting PUs from interference. Simulation results of MMAC-CR show high throughput and energy conservation. We refer the reader to [85, 86] for a detailed taxonomy and an updated literature review. The use of a CCC has its limitations. An in-band CCC is exposed to PU activities and thus its reliability and availability are not guaranteed while an out-of-band CCC requires either an extra radio interface or active switching between the frequency bands. In addition, control channel saturation can be a potential problem for a fixed common control channel [87]. Furthermore, a CCC needs to be allocated by a regulator and agreed through standardization. A recent development is the IEEE 802.19 working group [88], which focuses on the coexistence of different IEEE 802 technologies in the unlicensed spectrum. However, it will take a few years in reaching a consolidated agreement on standardization and commercially available solutions.
Simulation based studies have been conducted to show the effectiveness of the above mentioned protocols. However, these studies lack performance measurements in real environments and under realistic network conditions. Analytical models of protocols often make simplified assumptions in order to keep mathematical models tractable. Many of the simulation based studies are unable to model the physical layer accurately. Consequently, protocols often fail to provide the desired performance characteristics that have been predicted by analytical and simulation based studies [3, 86, 89].

To the best of our knowledge, C-MAC [90] is the first cognitive MAC which comes with a full prototype implementation. Instead of fixing a CCC, C-MAC uses a rendezvous channel, which is selected based on the reliability of available channels. It uses the TDMA principle, where each CR node has a distinct slot for periodic beaconing duration. This work is definitely a way forward and shows the feasibility of implementing a cognitive MAC. However, the evaluation of the prototype is not provided. Silvius et al. study the feasibility of a rendezvous based MAC protocol in CR paradigm in ad hoc environments on a testbed consisting of four nodes [91]. AMAC [92] is an adaptive MAC protocol for supporting MAC layer adaptation in cognitive radio networks. Based on the observation of network traffic changes, AMAC adapts between CSMA and TDMA MAC protocols at runtime. AMAC has been implemented on GNU radios on the ORBIT radio grid testbed [93] which shows the feasibility of implementing dynamic MAC switching in cognitive radio testbeds. The experimental results show slight throughput improvement in dynamic traffic environments.

O’Sullivan et al. have implemented a modified Aloha-based MAC protocol with implicit acknowledgements on USRP1 [94]. They aim at minimizing the intra-flow contention in multihop networks for achieving higher throughput and lower latency. They claim that an efficient implementation of a simple MAC protocol can potentially overcome the timing related issues on a low-cost USRP1 hardware. RAP-MAC [95] is implemented on WARP platform which performs packet based probabilistic rate and transmission adaptation. The commonly unused channel 14 of the 2.4 GHz band is selected as the CCC to ensure good channel quality in the conducted experiments. The experimental setup includes only one transmitter and receiver pair SUs. The scalability of the approach and the level of co-existence with other secondary networks have not been investigated. We have developed CogMAC [96], which is a decentralized Cognitive MAC protocol based on multi-channel preamble reservation scheme. The protocol dynamically selects and available communication channel using a distributed channel selection algorithm and allows nodes to be completely asynchronous to each other. CogMAC adaptively expands and contracts the number of frequency channels to be used depending on the network interference conditions. The protocol has been implemented on WARP boards and the results show high network packet delivery ratio with various interference patterns.
Although there has been numerous adaptive and flexible MAC protocols developed on either commodity hardware or research oriented SDR platforms for different types of wireless networks, the adaptation is mainly achieved by changing of parameters or switching among a limited number of pre-defined protocols. This kind of small-scale adaptation does not suffice the increasing demand of varying applications and complex spectral conditions. In this dissertation, we aim at providing a solution for generic runtime reconfiguration and flexible MAC realization for wireless networks, which allows a higher degree of adaptation flexibility as compared to existing solutions.

2.5 Component-based Design Approach for Flexible Protocols

In search of means to add flexibility to well established systems, many researches turned to component-based designs. Component-based design refers to the concept of decomposing a system into independent and simple units. The components should be observed as black boxes with defined inputs and outputs which can be developed and deployed independently [92–99]. While system-based design makes efforts on how to organize a structure on the global point of view, component-based design focuses on decomposed and decoupled units. Generally systems can be decomposed into components with different services and functionalities. With good encapsulation, final users only need to know the specific interfaces provided by these components to build applications. In addition, modification of a single constituent component in an application does not lead to an impact on the whole system. Therefore, the complexity in maintaining and upgrading the system is reduced. Many component-based platforms like Component Object Model (COM)/ Distributed COM [101], CORBA Component Model (CCM) [26] and Enterprise Java-Beans (EJB) [101] are widely used in the market. Such compositional design has many significant advantages. It gives systems a clearly visualizable structure. The system logic is explicitly shown due to the decomposition and encapsulation. Furthermore, components are reusable and extensible. New components can be developed and published by third parties, which facilitates the growth and the maturity of a system.

Component oriented protocol design [97] has been well investigated and many solutions have been proposed and implemented to allow flexibility and adaptability [102, 103]. The x-kernel is an architecture for composing network protocols [104]. An x-kernel configuration is a graph of processing nodes and packets are passed between nodes by virtual function calls. The x-kernel architecture is able to accommodate a wide variety of protocols with competitive performance to the traditional monolithic implementations. Netgraph [105] is a modular networking system. Netgraph allows on-line configuration modification by adding and/or deleting constituent nodes. Configurations can
2. Related Work

be built up dynamically. Click [10] is a software architecture for building flexible, configurable and extensible routers. It is a component-based system where the components are called elements which represents simple router functions. Like x-kernel, a router configuration is a directed graph with elements at the vertices and packets flow along the edges of the graph. Click router is able to achieve good performance with small overhead.

In the context for SDRs, both GNU radio and Iris are component-based designs which allow flexible protocol realization in software. Airblue [107] proposes a system where both the PHY and MAC layers are implemented in FPGA in a modular fashion to achieve low latency cross-layer communication to facilitate cross-layer wireless protocol development. Airblue is able to achieve comparable performance to the IEEE 802.11 commodity hardware. However, the focus of Airblue is not on providing flexible MAC designs and the modules provided by the system are not enough for complicated MAC realizations which are needed for networks to for example co-exist with others. WMP [38] and C-MAC [58] are tools for MAC protocol configuration using state-machine representations and decomposed MAC functionalities. They are designed targeting IEEE 802.11 and 802.15.4 standards, respectively. OpenRadio [108] proposes a programmable wireless dataplane which provides modular and declarative programming interfaces across the entire wireless stack. It shares similarities with software defined networks efforts such as OpenFlow [109] which focus on layer-3 of the network protocol stack.

In addition to flexibility, we see that a component-based design allow easy and fast protocol composition [106]. In the area of sensor networks, efforts have been made to ease the implementation burden and maximize code reuse by defining a unified protocol structure. A unified link layer abstraction is proposed to implement a broad range of networking and data link technologies without significant loss of efficiency [110]. A Unified Link-Layer API (ULLA) [111] offers a common interface to retrieve link layer information independent of the underlying hardware. In Unified Power Management Architecture (UPMA) [112] and its extension [113], power management features are separated from the core radio functionality to be easily plugged into different hardware specific MAC implementations. The component-based MAC Layer Architecture (MLA) [114, 115] further extends the UPMA idea by decomposing platform-independent part of MAC protocols into reusable components. Significant code reuse across different protocols with a low memory overhead and without significant loss in terms of performance metrics has been achieved. However, the hardware and protocol specific code which were not modularized still contributes heavily to the MAC implementation effort.

In this dissertation, we introduce a component-based architecture for fast and efficient MAC protocol realization. MLA is used as benchmark comparison in Chapter 3 and we have shown that our decomposition approach results in a higher level of code reuse. We have designed and implemented tools for runtime protocol configuration for two widely different types of networks:
cognitive radio networks and wireless sensor networks. COTS platforms are used for our implementations. Automated protocol optimization on-the-fly has been enabled together with many other features which are to be described in this thesis.
MAC protocols are responsible for efficient sharing of the communication medium among different nodes in a network. A wide range of applications and deployment conditions have imposed varying requirements on wireless networks and the QoS provided by the MAC layer. Since wireless technology, application requirements and network protocol standards are developing rapidly at the same time, the number of MAC protocols proposed is increasing tremendously. Although these MAC protocols exhibit vastly different performance characteristics, they essentially use the same set of functionalities of the underlying hardware to coordinate the communication access to the medium among transmitters and receivers. In this chapter, we identify and define a set of fundamental MAC functionalities as a library so that a wide range of MAC protocols can be easily realized by simply combining these functional blocks in an appropriate manner.

We have considered MAC protocols for wireless ad hoc networks, cognitive radio networks and sensor networks such as IEEE 802.11 [116], C-MAC [20], S-MAC [117], B-MAC [55], TRAMA [118], Z-MAC [54], etc. MAC protocols for other types of networks such as IEEE 802.15.1 [119] standard for wireless personal area networks, BSMA [120], BMW [121] and ADHOC-MAC [122] for broadcast Mobile Ad-Hoc Networks (MANETs) and Vehicular Ad-Hoc Networks (VANETs) can also be easily implemented using the same set of blocks. These blocks should be applicable to all hardware platforms and provide a hardware independent interface for MAC protocol realization. In order to verify the feasibility of the identified key MAC functional components, we have implemented these components on a list of platforms with different hardware characteristics. Sensor nodes such as TelosA, TelosB, MICA2 and MICAz from MEMSIC [123] are used as highly resource constraint platforms while WARP SDR boards are used for complex MAC algorithm realizations. A Java swing based Graphical User Interface (GUI) has been designed and implemented on sensor nodes to show how the MAC protocol implementation effort can be drastically reduced by using the component-based framework. The design and implementation of decomposable MAC framework has enabled a series of new functionalities in MAC layer realization and configuration which are described in Chapter 4, 5 and 6.

In this chapter, we identify the design goals of this framework in Section 3.1. In Section 3.2, we describe a list of identified essential building blocks.
with defined interfaces and parameters. Section 3.3 presents the details of our implementation of the blocks on sensor nodes and WARP platforms. A few sample MAC protocols realized on the hardware platforms using the defined blocks are presented as well. Evaluation results in terms of code re-use and memory consumption are discussed. Section 3.4 describes the GUI design for rapid MAC protocol realizations. The performance of the MAC protocol realized using our framework is compared to results obtained from the reference implementation. This chapter is mainly based on our articles [124–127], which were published during the dissertation work.

3.1 DESIGN GOALS

We list the major design goals for our MAC development framework in the following:

• **Fast Prototyping:** Due to the rapid development of wireless standards and technologies, a wide range of diverse applications have emerged. These applications often impose varying requirements to the network performance. The technology advancement and application development have jointly lead to the need for fast prototyping of different MAC protocols. MAC protocols have been typically hand coded targeting a specific platform using the platform specific language. The high implementation efforts and the lack of experimental support in the MAC development frameworks have been the major hurdles in prototyping new MAC algorithms in a rapid and efficient manner. A framework for fast realization of different MAC layers for varying wireless technologies has become very important for industrial and academic research community.

• **Portability and Code Reuse:** MAC protocol implementations for wireless networks have been often carried out in a monolithic fashion with tight coupling to the underlying hardware platform. Especially since MAC layer interacts with PHY layer closely and requires information from the radio front-end to operate, decoupling MAC layer operations from hardware platform can be complicated. This not only restricts portability across different platforms and the possibility of code reuse, but also the ease of code modifications. Therefore, it is important for a development framework to provide MAC protocol abstractions with well-defined interfaces, which allow code reuse across different MAC protocol implementations and facilitate portability.

• **Extensibility:** Due to the fast evolving technology in terms of computational power of wireless terminals, advancing spectrum access algorithms and support for high data rate transmission, new features of MAC protocols are demanded such as dynamic frequency selections, cooperative
3.1. Design Goals

New protocols are to be devised. Instead of designing and implementing entirely new protocols from scratch, extending the already existing ones can be a fast and efficient option. Therefore, in designing MAC protocols, one should also envision the future needs and devise an easily extensible architecture. If new feature arises, quick changes can be included in the MAC protocol development framework in a plug-in fashion to suit to the new challenges.

- **Runtime Reconfiguration:** One of the key characteristics of future MAC protocols will be the ability to adapt to the changing environment. Especially in the case of spectrum agile and cognitive MAC protocols, intelligent management of spectral resources, cooperation among nodes and advanced sensing in medium access procedures require adaptability and flexibility. Adaptability and reconfiguration based on network statistics and channel conditions for meeting the application QoS requirements are needed during the execution of a cognitive MAC protocol. We see a great potential in the spread usage of spectrum agile and cognitive MAC schemes. Therefore, the framework for MAC protocol development should provide possibilities of supporting mechanisms for fast on-the-fly reconfigurations of designed MAC protocols.

- **Hardware-software Co-design:** MAC protocols implemented in hardware devices such as IEEE 802.11 NICs, Bluetooth devices, etc. leave limited room for reconfiguration and customization. These static and rigid hardware based MAC implementations are inefficient for fulfilling the diverse and changing application demands and thus fail in flexible spectrum management domains [9]. Pure software implementations although provide full flexibility and offer reconfiguration capability at various levels, often remain incapable to meet the time critical deadlines. Therefore, it is necessary for software based MAC protocols designs to benefit from hardware accelerations. Hardware-software co-design approaches and hybrid processor FPGA designs such as Garp [128] becomes desirable.

- **Granular Parameters:** MAC protocols interact closely with PHY layer and radio front-end. Extended and fine-grained access control to different PHY and MAC parameters are needed, especially for spectrum agile cognitive networks [7]. Having the right level of granularity of exposed parameters allows easy control of MAC level functionalities without complicating the protocol implementation process by providing too much platform specific details. These parameters are needed to be exposed through a rich set of Application Programming Interfaces (APIs) in a MAC development framework.
3. LIBRARY OF REUSABLE COMPONENTS

Different MAC protocols share the same set of functional commonalities. We have analyzed a number of MAC protocols based on the CSMA, TDMA and hybrid principles in order to identify the basic common functional components. Using these basic components as building blocks, different MAC protocols can be realized. This component-based approach reduces the design and implementation complexity of protocols. The idea is similar to the Lego philosophy, where complex structures can be constructed using basic building blocks. A particular MAC protocol can then be easily composed by just connecting these unit components together as illustrated in Figure 3.1. This approach increases code reusability and maintainability. It enables efficient and fast MAC protocol realization and offers the possibility to configure MAC protocol effortlessly. In this section, we describe the list of basic functional blocks identified. All the components are associated with inputs, outputs and functions performed by the components.

Figure 3.1: Realization of two different MAC protocols using the same set of fundamental building blocks.

3.2.1 Timers

Timer is one of the most basic elements in a MAC protocol. Different degrees of precision are required depending on the nature of the task within a MAC protocol implementation. For instance, if a timer is used for TDMA slotting, the required precision typically is high, i.e. in microseconds. As the data rate required at the network level increases, some CSMA-based MAC protocols are starting to require microsecond class timer accuracy and granularity in order to support high data rate. On the other hand, if a timer is used for random back off or periodic beaconing, a precision to the tenth of a millisecond is usually sufficient. In short, timers are needed whenever the MAC protocol has to carry out actions at specified instances of time. Two common usages of timers include:
### Table 3.1: Timer component interface.

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>create()</td>
<td>creates a timer instance</td>
</tr>
<tr>
<td>Inputs:</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>{ONE_SHOT, PERIODIC}</td>
</tr>
<tr>
<td>Precision</td>
<td>{Millisecond, Microsecond}</td>
</tr>
<tr>
<td>Duration</td>
<td></td>
</tr>
<tr>
<td>Callback function signalled when the timer is fired</td>
<td></td>
</tr>
<tr>
<td>destroy()</td>
<td>remove a timer instance</td>
</tr>
<tr>
<td>start()</td>
<td>starts a timer</td>
</tr>
<tr>
<td>stop()</td>
<td>stops a timer if the timer is not expired yet</td>
</tr>
<tr>
<td>suspend()</td>
<td>suspends the running of a timer until it is resumed</td>
</tr>
<tr>
<td>resume()</td>
<td>resumes the running of a timer after the suspension</td>
</tr>
<tr>
<td>getStart()</td>
<td>returns start time of the timer</td>
</tr>
<tr>
<td>getDuration()</td>
<td>returns timer duration</td>
</tr>
<tr>
<td>getNow()</td>
<td>returns current time</td>
</tr>
<tr>
<td>getStatus()</td>
<td>returns the status {RUNNING, SUSPENDED, STOPPED}</td>
</tr>
</tbody>
</table>

- One-shot timers: Signal MAC layer when a time interval $t$ has elapsed from the current timestamp. For example, to realize a random backoff procedure in a CSMA-based MAC protocol, $t$ is often a random backoff time obtained from a random number generator.

- Periodic timers: Signal MAC layer at periodic time instants. Such operation is, for example, needed in TDMA-based MAC protocols where accessing the channel is confined to periodically occurring time intervals.

For both types of timers the basic operations are similar. One should be able to create a timer instance, associate the instance with properties such as the type and precision, duration and callback function when the timer is expired. The time duration should be modifiable and one should be able to suspend a running timer and resume its action afterwards. The capability of retrieving the current elapsed time from a timer is also desired. As an example, Table 3.1 lists the commands exposed in the Timer block.

#### 3.2.2 Checksums

MAC protocols almost always use checksums to verify the integrity of the received frame. Usually checksums are non-secure hash functions, such as cyclic redundancy check (CRC) codes of varying lengths. In security-related applications, cryptographic hash functions are used as well. The value of the check-
sum is computed over the entire or some specified part of the frame. Checksum bits are usually appended at the end of the frame. Since the number of checksum algorithms in use is large, the best approach should be to have parameterized checksum calculation routines for common cases. For example, in the case of CRC codes, configuration of such a block would involve setting the length of the code, the corresponding polynomial and the frame which the CRC should be calculated for. Longer CRC polynomial gives more reliability in detecting packet corruptions but leads to larger transmission overhead. Therefore, a CRC of flexible and even adaptable length is desired so that it can be selected based on the application requirements, channel conditions, etc. The interface defined for the checksum block is shown in Table 3.2 together with the rest of the basic blocks introduced in this section.

3.2.3 Carrier Sensing

For all CSMA-based MAC protocols carrier sensing is an operation of fundamental importance. Contention based protocols use carrier sensing to detect activities in the medium. For example, a sender relies on carrier sensing to decide whether or not to initiate data transmission. Carrier sensing is used to reduce collisions following the listen-before-talk principle. However, it does not guarantee 100% reliable transmission [129]. Carrier sensing is usually performed for a specified duration. The length of the duration should be flexible

<table>
<thead>
<tr>
<th>Block</th>
<th>Command</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checksums</td>
<td>getCrc()</td>
<td>CRC Length, Frame, Polynomial ID</td>
<td>CRC bits</td>
</tr>
<tr>
<td>Carrier Sensing</td>
<td>carrierSense()</td>
<td>Sensing Duration, Detection Threshold</td>
<td>Free/Busy</td>
</tr>
<tr>
<td>Radio Control</td>
<td>switchToState()</td>
<td>RX,TX,SLEEP, SLEEP</td>
<td>SUCCESS/FAIL Current State</td>
</tr>
<tr>
<td></td>
<td>getState()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Frequency Selection</td>
<td>setFrequency()</td>
<td>Frequency</td>
<td>SUCCESS/FAIL Operating Freq.</td>
</tr>
<tr>
<td></td>
<td>getFrequency()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transmission Power Selection</td>
<td>setTxPower()</td>
<td>Transmission Power</td>
<td>SUCCESS/FAIL Tx Power</td>
</tr>
<tr>
<td></td>
<td>getTxPower()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random Number Generator</td>
<td>getRand()</td>
<td>Range, Precision, Seed Value</td>
<td>Generated Num.</td>
</tr>
<tr>
<td>Send Frame</td>
<td>sendFrame()</td>
<td>Destination Address, Frame Type, Length, Pointer to Tx Buffer</td>
<td>SUCCESS/FAIL</td>
</tr>
<tr>
<td>Receive Frame</td>
<td>receiveFrame()</td>
<td>Pointer to Rx Buffer</td>
<td>VOID</td>
</tr>
</tbody>
</table>
and dependent on factors such as the required level of detection reliability and the nature of the sensing mechanisms. Depending on the PHY capability, carrier sensing mechanisms can range from simple implementation such as energy detection to complicated procedures including matched filter based preamble detection, feature detection, and signal classification of detected carrier, etc. The complicated sensing mechanisms are often desired for DSA and CRNs since the capability to identify PUs, SUs and other interferers is appreciated. Each of the possible techniques has a number of parameters, such as the energy detection threshold, that should be tunable. Correct setting of such thresholds usually also requires the MAC protocol to be able to query other characteristics of the current PHY configuration, such as the noise floor. In today’s hardware platform, carrier sensing is usually realized by energy detection, i.e. retrieve the energy level from the medium and compare to the indicated detection threshold. The threshold can be either pre-fixed or on-the-fly modified according to the noise floor [55].

3.2.4 Radio Control

MAC protocol requires control and access to the radio states. It needs to switch the radio to transmission state to be ready for packet transmission, reception state for carrier sensing and packet reception, and sleep state for energy conservation. The radio control functionality is tightly coupled with the underlying transceivers. Some radios offer more states apart from transmission, reception and sleep. For example, in Texas Instruments’ CC2420 radio chip, multiple sleep modes are available with different power consumption levels. These are realized by having independent controls over the voltage regulator and crystal oscillator of the radio chip. As a generic interface, states including transmit, receive and sleep are sufficient to describe most of the protocols. For protocols which requires special capabilities offered by the radio, an extension to the API can be easily realized.

3.2.5 Frequency Selection

Multichannel MAC protocols are one of the fast developing trends in MAC research. A lot of the wireless networks operate in ISM bands and do not have a dedicated channel for communication. Therefore, the ability to select a free channel within the available band is necessary to reduce collision and increase throughput. DSA is also a must in CRNs where nodes are expected to identify and utilize spectrum holes by dynamically access different frequencies. Dynamic Frequency Selection (DFS) is first proposed in IEEE 802.11h standard [130] and later on fully incorporated to the current 802.11 standard. DFS is mandated for 5 GHz in Europe so that WiFi nodes can change its operational frequency channel once radar activities is detected. In order to improve throughput, some MAC protocols require a combination of multiple channels
for single data transmissions. Therefore, the frequency selection function is being used often in recently proposed MAC protocols.

3.2.6 Transmission Power Selection

There are a group of MAC protocols which use transmission power control extensively to control the transmission range of the nodes and thus the topology of networks in order to improve network throughput, packet transmission reliability and optimize network lifetimes [131–133]. Transmit Power Control (TPC) is an IEEE 802.11 standard feature used to avoid interfering radar activities in the 5 GHz band in Europe [130]. Transmission power is commonly adjusted based on the distance between nodes or the current channel condition to maintain a constant bit error rate and thus goodput.

3.2.7 Random Number Generator

A number of common MAC functions, such as selecting backoff intervals for a backoff procedure in CSMA-based MAC protocols or deciding on if to transmit or not in a p-persistent MAC scheme, involve use of random numbers. Usually pseudorandom numbers are used, generated using a number of different algorithms. Seed value for any such algorithm should be specified by the MAC layer. For implementing encryption and security functions algorithms of cryptographic grade are required, but these might have excess computational overhead for typical MAC use. For general MAC protocol usage, a random number generator which generates numbers with uniform distribution within a given range is sufficient. The degree of precision for the number to be generated needs to be specified. For example, if the precision is set to be 1, the returned randomly generated numbers will be 1, 2, etc.; while if the precision is 0.1, the numbers will be 0.1, 1.2, etc.

3.2.8 Send Frame

The Send Frame block takes care of the formation of a MAC frame and pushes the bit/byte streams to the radio for transmission over the air. A MAC frame is constructed including the MAC header (destination address, source address, etc.), payload of the frame and the CRC bits which appends the frame. There are commonly used frame types in MAC protocols such as data packet, beacon, preamble frame, acknowledgement, request-to-send, clear-to-send, etc. To simplify the MAC realization and execution, the structure of these frames can be pre-defined and all fields can be filled in before the frame is being transmitted. Although the interface of the Send Frame block is generic across different platforms, the implementation of the block is highly dependent on the underlying hardware. For example, for a radio which provides packet level interface, the Send Frame block can be realized by filling appropriate contents to different sections of the packet. For a radio which provides bit/byte level
interface without support for synchronization between sender and receiver, preamble bits and a locking sequence need to be inserted. The hardware platform differences are hidden behind the component API.

3.2.9 Receive Frame

Similar to the Send Frame component, the Receive Frame component also resides very closely to the radio. It is typically a function triggered by a hardware interrupt, i.e. when a packet is received and successfully demodulated at the PHY layer. Depending on the radio, callback events can also be associated with reception of a packet with good header and bad payload, or a packet with bad header. These additional information can be used by the MAC protocol to evaluate the current channel condition. Receive Frame should not filter duplicate packets by default. The decision should be left to the MAC layer to make. Address recognition can be either as part of the Receive Frame or as an independent entity. The MAC address can be incorporated into the PHY header to enable fast packet recognition. Once the destination address contained in the header does not match the address of the receiver, the packet is discarded without being fully received to reduce resource waste. It also reduces the detrimental effects of exposed node and hidden node problems. On the other hand, address recognition can also be performed at the MAC layer after the entire frame so that the non-addressed frame can be made of use by the protocol e.g. to get the meta-data information etc. conveyed in the packet.

3.2.10 Secondary Level Components

The pattern in which various above mentioned basic blocks are connected can be identical across different MAC implementations. Therefore, we define secondary level MAC building blocks, which are composed of basic blocks and other secondary blocks. Secondary level components are platform independent as they are built on top of platform specific basic blocks and they can be defined customarily by the end user. For example, Send Preamble is a common feature for all preamble sampling-based protocols as a preamble of certain length needs to be transmitted before transmitting the data. The RTS-CTS-DATA-ACK four way handshake is another example of commonly used secondary level components which appears in most IEEE 802.11 DCF inspired MAC protocols.

Figure 3.2 illustrates the realization of Random Backoff block with the associated inputs, outputs and execution flow. The random backoff functionality is very common in CSMA/CA based protocols and consists of the basic blocks Timer, Carrier Sensing and Random Number Generator. There are two modes of the Random Backoff block: carrier sensing enabled and disabled. The Random Backoff block takes in a Boolean variable called CarrierSensing as one of the input parameters. We use the getRand() function of the Random
3. Decomposable MAC Framework

Figure 3.2: Realizing the Random Backoff block using the basic blocks.

Number Generator (RNG) block to generate the backoff duration. The output of the RNG block is fed into the Timer block to create a timer of ONE SHOT type with generated duration in milliseconds. `timerFired()` is the callback function that will be executed when the timer is fired. An ID is returned after the timer creation. The timer is started immediately after its creation, using the ID number as the identifier. When carrier sensing is disabled, i.e. `CarrierSensing` is set to be false, a Null waiting state is assumed till the timer expires and the block exits. The Carrier Sensing (CS) block is not executed in this case. When `CarrierSensing` is set to be true, carrier sensing is performed throughout the backoff procedure. The Threshold and Duration parameters are required for the CS block. Duration is commonly set to be the minimum possible time to retrieve a RSSI (Received Signal Strength Indicator) value to be compared to the Threshold. If CS returns true, which means that the channel is busy, the timer is suspended and CS is performed again. The backoff timer is going to be suspended until the channel is free. Therefore, when the output `CS_out` is false, which indicates the channel to be free, the backoff timer is resumed if it was already suspended and CS is repeated.
Table 3.3: Commonly used secondary blocks.

<table>
<thead>
<tr>
<th>Block</th>
<th>Usage and the composition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Backoff</td>
<td>Random backoff mechanism&lt;br&gt;Timer, Random Number Generator, Carrier Sensing</td>
</tr>
<tr>
<td>Expecting Frame</td>
<td>Used when the node is anticipating a packet&lt;br&gt;Receive Frame, Timer, Radio Control</td>
</tr>
<tr>
<td>Send Packet</td>
<td>Called after seizing a channel free&lt;br&gt;Send Frame, Random backoff, Radio Control</td>
</tr>
<tr>
<td>RTS/CTS/DATA/ACK</td>
<td>Four-way handshake mechanism&lt;br&gt;Send Packet, Expecting Frame</td>
</tr>
<tr>
<td>LplCoordinator</td>
<td>Controlling the Low Power Listening (LPL) operation&lt;br&gt;Radio Control, Carrier Sensing, Timer</td>
</tr>
<tr>
<td>Send Preamble</td>
<td>Used in preamble-sampling based MAC protocols&lt;br&gt;Send Frame, Timer</td>
</tr>
</tbody>
</table>

When the timer expires, the backoff procedure is done and the block exits. If CarrierSensing is enabled for the random backoff block, the channel is free when the block exits. The figure shows a detailed control flow (sequence of execution and preemptions) in the RNG block. We have realized different secondary MAC blocks in our framework based on the same principle. Table 3.3 lists the most common of these blocks. The secondary level blocks can also be used as sub-blocks for other complex blocks.

3.2.11 Block Diagrams for Sample MAC Protocols

Figure 3.3 illustrates the realization of IEEE 802.11 DCF and S-MAC using the fundamental and secondary components as described Section 3.2. The details on data and control flow are omitted to keep the figure simple. The binding logic, decisions and variable assignments, which determine the state-machine of the MAC protocols are indicated through unfilled blocks. While the two protocols are designed with different level of complexity, they are realized using the same set of components. It is also worth noting that certain components, for instance Send Packet, Timer and Radio Control, are also used repetitively within the MAC realizations. In addition to the flowchart representation, we have implemented S-MAC on sensor nodes and IEEE 802.11 DCF on WARP SDR boards using our Decomposable MAC Framework (cf. Section 3.3).
3. Decomposable MAC Framework

Figure 3.3: Realization of IEEE 802.11 DCF and S-MAC based on the fundamental and secondary components.
In order to show the generality of the identified components for cross-platform portability, we have implemented the same set of components on two types of platforms. WARP SDR boards \cite{66} are used extensively for our research on reconfigurable MAC protocols due to its powerful FPGA and the opportunity offered for hardware-software co-design in realization of a flexible yet efficient MAC protocol. Our framework on WARP boards is implemented using both the OFDM Reference Design v.14 and v.16 \cite{135}. The implementation methodology and evaluation results are presented in Section 3.3.1. The MAC performance of two different types of MAC protocols realized on WARP boards using Decomposable MAC Framework is presented in Section 3.3.2. Sensor node platforms are typically characterized by its limited memory and battery resource. We have implemented all the basic components and several secondary components which some are popularly used in MAC protocols for WSNs such as Send Preamble and LPLCoordinator. The details of the implementation and evaluation results are presented in Section 3.3.3.

### 3.3.1 Decomposable MAC Framework Implementation for WARP Boards

We have implemented the identified MAC components described in Section 3.2 for WARP SDR boards. All the fundamental MAC components in the framework are implemented in the FPGA, except RNG, which is implemented in software on the PowerPC core. The virtualization of the fundamental components through flexible wrapper APIs is carried out in software running on the PowerPC processor. Most of the hardware related PHY and radio functionalities are provided by the OFDM Reference Design. We have extracted the functions required and provides a list of interfaces for the MAC components. The evaluation results presented in this section are obtained using the implementation based on version 14 of the OFDM reference design. We have later ported our framework implementation to the more updated version 16 of the reference design.

In our framework, we have exposed radio functionalities such as setting the transmit power levels, channel selection, radio state switching, etc. Some extended functionalities such as receiver sensitivity thresholds, setting modulation schemes and coding rate, etc. are also exposed in the same manner since these additional interfaces facilitate cross-layer interactions. For instance, our implementation of CogMAC \cite{96} using the framework lavishes from such a close PHY-MAC interaction. In order to avoid polling delays, we have modified the basic hardware reference design to include an interrupt controller for all the components available to the framework. Events such as the reception of a packet or CRC failure, are combined with user-defined Interrupt Service Routine (ISR) functions. The interface for each object is standardized...
to ease the debugging process. Particular hardware requirements include, for instance, granularity of timers or type of modulation schemes, etc., which may be defined on initialization or changed at the runtime, when required.

Code portability and ease of MAC designing are achieved through our high-level hardware abstraction architecture as shown in Figure 3.4. More complex MAC designs can be envisioned on the platform, where the number of required resources of a particular type exceeds the number of hardware components. The framework user is in this case exempted from the resource management tasks as the framework keeps state information upon freeing the resources. Timers are a typical example of such scheduled objects. In the User MAC Code, timer objects may be generated by specifying object properties - a timer may either be periodic or fires only once, the granularity of the timer is in the scale of milli- or microseconds. The Framework Object Interface manages those objects and destroys them upon expiration of their lifetime. Instead of assigning a particular hardware timer on the WARP board, the Virtual Component Code only updates the Object State Information. For this reason, an update to the duration of the timer object causes an update in the object state information. Only if inherently hardware-dependent operations such as starting the timer are requested by the user, the hardware scheduler allocates real resources. Interaction with those resources is carried out through platform-specific drivers that interact with the available hardware. Therefore, the MAC protocol designer using the Timer block can create and initialize as many timers as needed without having to be concerned with amount of hardware timers available. However, the current implementation limits the number of concurrently executing hardware timers to be seven which is the number of timers available from the WARP board. It is due to the fact for MAC protocol design it is very unlikely having to use more than seven timers at the same time. If it becomes a problem, it can be easily solved by building multiple software timers from a master hardware timer.
3.3. **Framework Implementation and Evaluation Results**

Table 3.4: Block reuse for MAC protocol realizations on WARP board.

<table>
<thead>
<tr>
<th>Block</th>
<th>Aloha</th>
<th>CSMA</th>
<th>B-MAC</th>
<th>IEEE 802.11</th>
<th>S-MAC</th>
<th>CogMAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send Packet</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Expect Frame</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>RTS/CTS/DATA/ACK</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Random No. Gen.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Timer</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Carrier Sensing</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Frequency Selection</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Radio Control</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Send Frame</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Receive Frame</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Component Reuse on WARP Board**

Table 5.4 lists the MAC block reuse in realizing various protocols on the WARP boards. It can be observed that certain blocks are used multiple times within a particular protocol, which advocates the idea of realizing the key atomic functionalities in the hardware for higher computing and communication efficiency. From the perspective of a user, a MAC protocol becomes simpler to implement, especially if the framework provides support for secondary level blocks. We have realized IEEE 802.11 DCF which is the industrial standard for WLAN, and CogMAC, a decentralized spectrum agile cognitive MAC protocol with fairly complex multichannel operation in a fast and easy way using the framework.

**Software versus Hardware Implementation**

We argue that implementing a MAC protocol purely in software fails to achieve timing requirements and therefore results in poor performance characteristics. On the contrary, a hardware implemented MAC protocol, e.g., IEEE 802.11 on a COTS NIC, though optimized for the design, does not permit flexible experimental room and the needed PHY-MAC interaction. We approach this problem by implementing selective basic functionalities in hardware while providing flexible APIs and virtualization in the software. This enables MAC protocols to simultaneously achieve hardware acceleration without compromising on flexibility. A one-to-one comparison in terms of the response time and execution speed (which directly affects the latency and throughput as shown in [36]) of a MAC implementation on a WARP board with an implementation on a standard IEEE 802.11 NIC or with a GNU radio based software implementation is unfair. Therefore, in order to study the benefits of implementing
the key atomic functionalities in hardware, as examples, we carried out the implementation of CRC and RNG blocks both in a custom hardware as well as in the software on an ARM926 core. We have observed a performance gain of ca. 2041% and 779% in the speed of the CRC computation and a random number generation through the hardware implementation, respectively.

3.3.2 Sample MAC Protocol Realizations on WARP Board using Decomposable MAC Framework

We have used Decomposable MAC Framework on WARP boards extensively in our research works on MAC algorithms for both classic and emerging network structures. In this section, we describe briefly two instances of MAC protocol implementations, namely IEEE 802.11 DCF and CogMAC, to show the versatility of our framework. Some selected measurement results are presented for each protocol. OFDM reference design v.16 is used for Decomposable MAC Framework in the implementations of these two protocols.

IEEE 802.11-like MAC Protocol Implementation

Although our MAC protocol implementation is not able to achieve throughput offered by IEEE 802.11 NIC, we have realized MAC features defined in IEEE 802.11 standards for DCF including physical carrier sensing, virtual carrier sensing, binary exponential backoff, RTS-CTS handshake, frame aggregations and block acknowledgement. Our implementation shows the capability of our framework in supporting feature-wise industrial standard MAC protocols.

Figure 3.5a and Figure 3.5b show the maximum goodput achieved by our implementation based on Decomposable MAC Framework, without RTS-CTS mechanism and with RTS-CTS, respectively. We have used one transmitter and receiver pair for our experiment. The antenna of the two WARP boards are placed close together to minimize the transmission loss. We have used minimum contention window size 15 and maximum contention window size 255. Binary Phase Shift Keying (BPSK) is used as the base modulation scheme which gives a 6 Mbps basic rate. We have applied BPSK, Quadrature Phase Shift Keying (QPSK) and 16-Quadrature Amplitude Modulation (QAM) with different coding rate to the packet payload of 1500 bytes. There has been numerous studies on 802.11 MAC DCF performance [136,137] in terms of mathematical modelling and simulations. We compare our results to the results presented by Mahasukhon et al. [138]. Although the results we are comparing against are not empirical measurements measured using any COTS WLAN card and access point, the authors have presented results on MAC layer throughput with one transmitter with different modulation schemes (cf. Figure 3 and Figure 4 of [138]) which is the same as our measurement settings. Simulation results often give a better result as compared to empirical measurements especially if a perfect channel is assumed. Naturally there are differences in the parameter
3.3. Framework Implementation and Evaluation Results

Figure 3.5: Goodput of one transmitter and receiver pair exercising 802.11-like MAC protocol. The packet size is 1500 byte.
settings between our implementation and simulation setup of our reference results. We focus on the trend of the MAC protocol performance instead of absolute numbers. We see that our measurement results are comparable to the simulation results for 802.11g with RTS-CTS mechanism though the basic rate used for 802.11g is only 1 Mbps. We do observe similar trend among these two sets of results. In terms of the maximum goodput achieved, the difference between MAC protocols with and without RTS-CTS mechanisms increases from 0.2 Mbps to 3.5 Mbps as the data rate increases. This observation is very similar to the results presented in [138].

Figure 3.6a shows the aggregated goodput of a network exercising basic 802.11 MAC procedures with RTS-CTS mechanism. The network has a star topology where there is one receiver and multiple number of transmitters. The packet generation rate is varied to create channel saturation. The packet payload size used in this experiment is 1500 byte. QPSK modulation scheme with 3/4 code rate is applied to all data traffic. We see that the maximum goodput is achieved with single transmitter. When more than one user joins the network, the maximum goodput has decreased by approximately 10%, possibly due to inevitable collisions and backoff durations. Figure 3.6b shows the fairness of resource sharing among nodes within the network under the same experimental condition. Jain’s fairness index [139] is a commonly used quantitative measure for resource allocation. The fairness index lies between 0 and 1 where 1 means absolute fairness. Equation 3.1 states how the index is calculated where \( n \) is the number of users and \( x_i \) is the throughput for the \( i^{th} \) connection.

\[
f = \frac{\left( \sum_{i=1}^{n} x_i \right)^2}{n \sum_{i=1}^{n} x_i^2}
\]  

(3.1)

In Figure 3.6b we see that the fairness index remains at 1 when the network is unsaturated with low packet generation rate. The index varies between 0.982 and 0.999 as the network becomes saturated. It indicates a high level of fairness among resource sharing using the contention scheme in 802.11 MAC protocol.

Figure 3.7 shows the maximum goodput of a single flow with RTS-CTS enabled at different packet sizes. In 802.11 legacy DCF MAC layer, one single packet transmission is accompanied by a significant overhead including packet header and tail added at each network layer, DIFS (DCF Interframe Space) and SIFS (Short Interframe Space) deferral, backoff, RTS-CTS exchange, ACK transmission, etc. Therefore, as the packet payload decreases, the overhead becomes significant. In Figure 3.7 we see that when the packet payload size is 10 byte, the maximum goodput is as little as 1% of the goodput achieved when the packet size is 1500 byte.

Since there are many small packets such as TCP acknowledgements within the network, ways to decrease the per packet overhead are desired. In order to improve throughput performance with small packet transmission, frame aggregation mechanism is introduced in IEEE 802.11n standard [140]. There are
3.3. Framework Implementation and Evaluation Results

Figure 3.6: Performance in terms of a) aggregated goodput and b) fairness of a network exercising basic 802.11-like MAC protocol with RTS-CTS mechanism. The network is in a star topology with varying number of transmitters.
two types of frame aggregation: aggregate MAC service data unit (A-MSDU) and aggregate MAC protocol data unit (A-MPDU). A-MSDU allows multiple MSDUs to be sent to the same receiver concatenated in a single MPDU. It means that one MAC header will be added to the concatenated packets and one frame check sequence (FCS) will be computed and appended to the concatenated packets. Only one acknowledgement is generated for one A-MSDU. If any subframe within a A-MSDU is corrupted, the whole A-MSDU needs to be retransmitted. An A-MSDU packet is transmitted when either the packet size threshold has been reached or when the maximal delay of the oldest packet reaches a pre-assigned value. A-MPDU joins multiple MPDU frames, i.e. each MPDU frame has its own MAC header and tail. Each MPDU frame requires to be acknowledged individually. Since multiple acknowledgements add to system overhead as well, block acknowledgement is used for A-MPDU. After transmission of a A-MPDU, a Block Acknowledgement Request (BAR) is sent to the receiver. The receiver replies with a Block Acknowledgement (BA) including an 8 byte bitmap indicating the packets which have been received. Each bit of the bitmap represents one MPDU within the A-MPDU and the maximum number of MPDUs allowed to be aggregated in one A-MPDU is therefore 64. By using the bitmap, the corruption of any individual frame within an A-MPDU can be indicated and retransmitted separately from the rest of the A-MPDU. A two-level frame aggregation combining A-MPDU and A-
MSDU can be used to further enhance the MAC performance. In a two-level frame aggregation, an A-MSDU within the A-MPDU frame size threshold is treated as a MPDU which can be transmitted together with other MPDU and A-MSDU frames as an A-MPDU.

We have implemented frame aggregation and block acknowledgement using Decomposable MAC Framework. Mathematical and simulation studies have been performed for the MAC enhancement strategies proposed in 802.11n standard [141, 142]. In our implementation, the maximum A-MSDU threshold is 2020 byte. This number is determined by the PHY layer transmitter and receiver buffer size of the ODFM reference of WARP board. We use a maximum number of 16 MPDUs for A-MPDU.

Figure 3.8 shows the maximum goodput achieved by a transmitter receiver pair with different packet payload sizes using various frame aggregation mechanisms. With A-MSDU aggregation, one RTS-CTS exchange is performed for each A-MSDU. For A-MPDU, one RTS-CTS exchange used for 16 consecutive MPDU transmissions, followed by an exchange of BAR-BA. We see that by using a two-level frame aggregation, the maximum goodput has increased from 5.6 Mbps to 7.7 Mbps. A-MSDU performs better than A-MPDU when the packet size is less than 700 bytes due to a larger overhead per packet with small packet sized MPDUs. The goodput at 1500 byte packet size is slightly lower than smaller packet sizes for A-MSDU and two-level aggregation. This is due
to the 2020 byte size limit for A-MSDU. When packets with 1500 byte size are constantly generated, they cannot be aggregated using A-MSDU; while when packets with 10, 20, 100, 500, 1000, etc., bytes are generated, they can be aggregated to form an A-MSDU of 2000 byte payload size.

Spectrum Agile MAC Protocol Implementation - CogMAC

Cognitive MAC protocols are designed to efficiently utilize the scarce spectral resources without affecting the performance characteristics of primary users. The use of spectrum opportunities can often require stochastic approaches due to difficulty in predicting their appearance. Infrastructure based coordinated access techniques are not viable for many applications and spectrum bands, especially in the case of wireless local area and sensor networks. We have designed and implemented CogMAC, a decentralized cognitive MAC protocol, which is based on the multichannel preamble reservation scheme. CogMAC protocol is targeted for infrastructureless environments. Spectrum agile MAC protocols often require extensive control of the PHY and radio functionalities for sensing and frequency hopping. CogMAC has been realized using our Decomposable MAC framework on WARP boards which provides the control knobs required by this protocol genre.

CogMAC uses a distributed channel selection strategy, which allows it to handle network and spectral dynamics in an efficient manner. The protocol uses the multichannel carrier sensing principle where a node sequentially scans all the channels in its channel pool to detect a transmission activity. Channel pool refers to a list of channels which a node can use for packet exchange. CogMAC protocol uses a heuristics based method for channel selection similar to the approach that is described in [143]. In every sensing cycle, each node scans channels in its pool sequentially for a potential activity. Weights are associated with each channel and are updated in every sensing cycle based on the type of the activity detected. If a particular channel is found free or data communication is established, the weight associated with the channel is increased. On the contrary, if a channel is found interfering or noisy, its weight is decreased. Channel weight history is also maintained, which helps in identifying poor quality channels and blacklisting them. Since single packet transmission is carried out with a significant overhead in scanning multiple channels and reserving transmission channels by extended duration of packet transmission, CogMAC allows multiple packets to be transmitted with a single channel reservation. A detailed description of the MAC protocol design can be found in [96]. In this section, we are presenting selected results on the effect of the number of transmitters and the traffic generation rate on the network goodput and we compare our channel selection algorithm to a random channel selection scheme.

We have carried out an experiment to study the effect of increasing traffic in a network without an interferer and in the presence of a cyclic interferer.
The cyclic interferer sequentially hops to different channels with a dwell interval of 100 ms. In this experiment, we considered a channel pool size of two, a fixed packet size of 1000 bytes, a fixed carrier sensing duration of 30 µs, and a PU allowance time of 15 ms. A larger traffic volume was obtained by generating higher number of packets at a transmitter and by increasing the number of transmitters in the network in a star topology. Figure 3.9 shows that with faster traffic generation rates at a transmitter, the packets received at the receiver go up in an exponential manner. It is worth mentioning here that CogMAC exercises its own adaptive multipacket scheme depending upon the traffic volume (the number of packets queued at a transmitter) at a given time. The achieved goodput is slightly higher when no interferer is present. A small difference indicates that CogMAC is effectively able to dynamically avoid the interferer. Introducing more transmitters lead to an overall higher traffic in the network and therefore the achieved goodput at the receiver also show a corresponding increase. The effect is more clear at lower traffic generation rates while at higher traffic generation rates, the network tends to attain saturation.
Figure 3.10: Achieved goodput with respect to the increasing traffic in the network in the presence of a cyclic interferer with a dwell interval of 100 ms. The traffic is increased by increasing the packet generation rate at a node and by increasing the number of transmitters in a star topology. A fixed packet size of 1000 bytes, a channel pool size of 2 and 4, a carrier sensing duration of 30 $\mu$s and a PU allowance time of 15 ms are considered.

Figure 3.11 shows the experimental results on the effect of the channel pool size on the achieved goodput with increasing network traffic. The same network setup has been used as in Figure 3.9. We see that the achieved goodput increases with the increasing number of transmitters and the traffic generation rate. It is due to the fact that the channels are not saturated with the traffic generated. The obtained goodput falls with an increasing channel pool size, which is as expected since as the number of channels increases, the overhead per packet transmission increases.

Figure 3.11 shows the comparison of a random channel selection scheme with CogMAC in the presence of an interferer that cyclically sweeps frequency channels with a dwell time of 100 ms in the channels used by the WARP boards. It can be observed from the figure that CogMAC is able to achieve significantly higher goodput compared to the random channel selection method. In these experiments, instead of using the heuristics for channel selection, a channel is selected randomly from the pool in the case of RandomMAC. The achieved goodput goes down linearly with an increasing number of channels in the
3.3. Framework Implementation and Evaluation Results

**Figure 3.11:** Achieved goodput comparison of CogMAC with a random channel selection scheme in the presence of a cyclic interferer with a dwell interval of 100 ms. A fixed packet size of 1000 bytes, a carrier sensing duration of 30 $\mu$s and a PU allowance time of 15 ms are considered.

A carrier sensing duration of 30 $\mu$s and a fixed packet size of 1000 bytes was used in this experiment. Figure 3.12 shows the corresponding packet delivery ratio. We can observe that CogMAC is able to achieve ca. 100% packet delivery ratio while the packet delivery ratio for random channel scheme stays much lower. While the wireless channel cannot be 100% reliable and factors like channel fading, inherent noise or the transceiver decoding errors may lead to packet losses, we achieve a high success rate in transmission. This result is explained by a good forward error control and excellent channel conditions, helped by the MAC itself that minimizes the interference between radios (in the case of small number of nodes this gain is very large). Please also note that at a channel pool size of two, RandomMAC gives much higher successful delivery ratio compared to larger pool sizes. This is because of the CSMA property of the MAC scheme, i.e., even with the random channel selection, a packet transmission is initiated only if the channel is found to be available. However, in the case of RandomMAC, inability to ensure the availability of the channel during the packet transmission results in packet losses due to interference.
3. Decomposable MAC Framework

Similarly, we have implemented the components described in Section 3.2 for sensor nodes in TinyOS 2.x. As an example for a commonly used secondary block in preamble sampling based MAC protocols, we describe the Send Preamble component in details.

LPL is a strategy used by MAC protocols to duty cycle the radio by sampling channel periodically to save energy consumption. In LPL, most of the time the receiver’s radio is in sleep state. For data communication to take place, both the transmitter and the receiver have to be awake at the same time. In an unsynchronized and decentralized network, the transmitter and receiver are not aware of each other’s transmission/wake-up schedule. In preamble sampling based MAC protocols, the transmitter has to wake the receiver by sending a long preamble before sending the actual data. There are several techniques in sending a preamble designed by various MAC protocols. For example, over a packetized radio, B-MAC [55] broadcasts repeated back-to-back preamble frames for a fixed duration to wake the receiver. The duration is directly related to the duty cycle of the receiver. MFP-MAC [45] unicasts...
3.3. Framework Implementation and Evaluation Results

preamble frames with a small gap between them. Similarly, X-MAC [16] unicasts preamble frames and waits for an acknowledgement from the receiver after each preamble frame. We have designed and implemented a generic component Send Preamble which encapsulates preamble related functionality and takes into account the complexities of different preamble sampling protocols. The design of Send Preamble is flexible and could be easily configured according to the needs of various preamble sampling MAC protocols. It is composed of two basic components, i.e. Send Frame and Timer. The Send Preamble component has command:

```c
void sendPreamble (message_t* msg, uint8_t len,
                   uint16_t preambleLength, uint8_t preambleType);
```

Attribute `preambleLength` is the specified preamble length interval in milliseconds. Within this time interval, Send Preamble component will send preamble frames repetitively using the Send Frame block. Send Preamble component can send a monolithic preamble, micro frame preamble, data frame preamble or a short preamble depending on the specified `preambleType` attribute, i.e. MONOLITHIC, MFP (Micro Frame Preamble), DFP (Data Frame Preamble) and STROBED, respectively. A Timer component is used to govern the preamble frame transmission duration. It triggers the call-back event `preambleSendDone()` when the timer has timed out.

We have implemented Aloha, simple CSMA, B-MAC, S-MAC, X-MAC and MFP-MAC using the component library. We have designed and implemented MAC-PD, which offers a graphical user interface for MAC design and automatically generates nesC code from the graphical design. The details of the protocol designer is to be introduced in Section 3.4. The results we present in this section are based on the auto-generated TinyOS code through MAC-PD. Figure 3.13 shows a simplified component diagram of MFP-MAC implementation. We can see that the low-level platform specific drivers have been encapsulated by basic level components. User designed components are then built upon the basic level components and other higher level components.

In this section, we evaluate the component and code re-usage when using our framework for a number of MAC protocol implementations. Table 3.5 lists the components which are used in the implemented protocols. Both the basic level blocks and the secondary level components are included in the Table. It may be noted that Radio Control, Send Frame, and Receive Frame blocks are fundamental to all the protocol realizations. Timer instances and carrier sensing functionalities are also common to all the contention based protocols. Having a rich set of reusable components reduces the implementation and debugging efforts on part of the MAC designer since one can reuse the already tested components for new protocol realizations. The implementation effort reduction also indicated through a high proportion of the reusable lines of code.
Figure 3.13: Simplified component diagram for MFP-MAC protocol on TinyOS.

Figure 3.14 shows the total number of lines of nesC code and the number of lines of reusable code for different MAC protocols implemented through our framework. Please note that the nesC code is completely auto-generated through MAC-PD and here the unused portion of the code constitutes the binding logic for different components and the protocol specifics. It is remarkable that the overall proportion of the code reusability across different MAC implementations is approximately 80%. This indicates that our framework gives a high degree of code reusability and provides a fast means for prototyping different MAC protocols.

Figure 3.15 compares the proportion of the nesC code reuse for our framework to MLA [114] approach. MLA is a component-based MAC layer ar-
Table 3.5: Component reuse for MAC protocol realizations.

<table>
<thead>
<tr>
<th>Block</th>
<th>B-MAC</th>
<th>X-MAC</th>
<th>MFP-MAC</th>
<th>S-MAC</th>
<th>Aloha</th>
<th>CSMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Send Preamble</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Expect Frame</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Send Packet</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BEB</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>LplCoordinator</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Send Frame</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Receive Frame</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Radio Control</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Timer/Alarm</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Rand. No. Gen.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Carrier Sensing</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Architecture for WSNs which we have introduced in Section 2.3. It is used as a benchmark to evaluate the effectiveness of our decomposition approach. It is evident that our approach allows a higher proportion of code reuse as compared to the MLA. For the three protocols considered, the code reuse for our framework is above 80% while it is approximately 55% for MLA. The total number of lines of code for the MAC implementations in MLA is approxima-
Figure 3.15: Comparison of the overall number of lines of code and the percentage of the reusable code between MLA and MAC-PD.

3. Decomposable MAC Framework

tely 75% of our implementations. The bigger total number of lines of code is due to the different level of component granularity and higher component modularization. It is one of the trade-offs we see for the flexibility and the code reuse.

Memory Footprint

The generated MAC code is embedded with the rest of the components and with the TinyOS 2.x scheduler as a single executable binary. Table 3.6 lists

Table 3.6: Memory footprints [byte] for MAC protocols on TelosB node.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>TinyOS 2.0.1</th>
<th>TinyOS 2.1.1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RAM</td>
<td>ROM</td>
</tr>
<tr>
<td>B-MAC</td>
<td>758</td>
<td>18074</td>
</tr>
<tr>
<td>X-MAC</td>
<td>690</td>
<td>17660</td>
</tr>
<tr>
<td>CSMA</td>
<td>632</td>
<td>14026</td>
</tr>
<tr>
<td>S-MAC</td>
<td>724</td>
<td>15188</td>
</tr>
<tr>
<td>MFP-MAC</td>
<td>758</td>
<td>18074</td>
</tr>
<tr>
<td>Aloha</td>
<td>610</td>
<td>13700</td>
</tr>
</tbody>
</table>
the memory footprint in terms of RAM (Random-Access Memory) and ROM (Read-Only Memory) consumption for the binary. Please note that the ROM usage represents the actual code size while the RAM size indicates the variables stored in the memory. The table shows the effect of CC2420 driver stack implementation in TinyOS 2.0.1 and TinyOS 2.1.1 for TelosB platform. It can be observed that there is a slight decrease in the code memory in the later version because of more modularized stack implementation. Table 3.7 compares the memory footprints on TelosA, MICAz and Intelmote2. It is worth noting that the TinyOS 2.x binary along with the MAC protocols occupies less than approximately 5% of the available RAM and ROM on these platforms.

Table 3.8 shows the comparison of the memory footprints for different MAC implementation among the monolithic implementation, MLA based implementation and the implementation using our framework (labelled as MAC-PD). In order to ensure fairness in our analysis, we used the same CC2420 driver stack (TinyOS 2.0.1) and the same test application for the three design approaches. The table shows that the MAC implementations using our framework are clearly more memory efficient than MLA and on-par with the monolithic implementations in terms of B-MAC and X-MAC implementation. The memory consumption overhead for our approach is more significant for a simple CSMA MAC procedure due to the low complexity of the MAC itself.
3.4 MAC Protocol Designer

In an effort to enable easy MAC protocol design and implementation, we have developed a tool - MAC Protocol Designer (MAC-PD) which has an interactive graphical user interface for rapid MAC protocol prototyping. MAC-PD allows designers to express MAC protocols in the form of flowcharts composed of various basic MAC building blocks as defined in Decomposable MAC Framework. It is able to directly generate the executable target code for different platforms without requiring code development efforts. It takes off the burden of learning programming language and platform details from the designers and eases the MAC designing process through an interactive drag-and-drop based user friendly environment. In this section, we are describing MAC-PD used for a wide range of commonly used sensor node platforms such as TelosA, TelosB, MICAz, BSNnode [144] and IntelMote2 [145]. MAC-PD is able to generate TinyOS 2.x source code (nesC) [146] and executable scripts, which are directly compiled and downloaded onto above listed COTS sensor node platforms. XML model files generated from the flowcharts are used for storing the re-loadable design developed by a user. The user can thereby simply reuse and modify previously saved sessions. The user can also add a particular customized design to the library as a reusable component, which reduces the future designing efforts. Figure 3.16 shows the layered architecture of the framework, where each layer performs its tasks in a top-down work flow. The layers are decoupled through well-defined interfaces. The details of each layer is presented in subsections later. We have also applied the similar drag-and-drop graphical interface for generic MAC protocol design as part of our toolchain for runtime protocol realization to be presented in Chapter 4.

![Architecture of MAC-PD for auto code generation based on Decomposable MAC Framework.](image)
3.4. MAC PROTOCOL DESIGNER

3.4.1 User Interface

The top most layer provides a flexible and user-friendly interface for MAC designing. It is based on the Java-Swing GUI (cf. Figure 3.17) and allows interactive “drag-and-drop” feature for MAC development. All the basic and custom defined MAC components are made available to the user in the GUI. A designer can simply select a component and drag it to the provided drawing grid. The User Interface layer gathers the information of the state-machine and logic flow of the MAC design from the flowcharts.

Figure 3.18 represents the graph data structure used for maintaining a particular block design. There are two types of edges: ContainsEdge and FollowEdge. ContainsEdge reflects the implementation of a method. It specifies a parent and child relationship between nodes. For example, an IF-ELSE node can have outgoing ContainsEdges linking to the nodes to be executed when condition of the conditional block is satisfied. FollowEdge is used for showing the connection among the nodes at the same level. Four different types of nodes are used to maintain the states of different TinyOS 2.x constructs. BlockNode maintains all the information related to method calls of the TinyOS 2.x specific constructs whileDeclarationNode keeps track of the state information of different declarations. The nodes in the graph represent TinyOS 2.x command calls, event signals, function calls, task postings and the corresponding declarations, event-handlers and implementations. Furthermore, the program logic with IF-ELSE statements and arithmetic, Boolean and logical expressions are represented through the IfElseNode and Expres-
A user sets the states of the nodes through GUI and defines connections among the nodes to complete MAC design. The edges in the graph represent the flow of a protocol. While parsing, the top layer processes the information of the graph and reports the missing or invalid data. It also validates the basic design and reports illogical connections.

3.4.2 XML Modelling

This layer is responsible for maintaining an XML model of the user design. XML tags are generated for each node, which preserves the state of a node and its relationship with other nodes in the graph. XML model is used to store TinyOS 2.x constructs in a tree structure. All the user designs are preserved in XML so that these can be utilized and updated without having the need to redesign everything from scratch. For instance, we implemented MFP-MAC simply by few modifications in the flowchart design of B-MAC in the GUI. Figure 3.19 shows the XML definition of the Radio Control component as an example. It contains the nesC module, configuration and interface file listings and the definitions of all the declarations associated with the interface. The figure shows only the `startRadio` command. All the five possible declarations in nesC, commands, tasks, event definitions, event handlers and normal functions [147], are modelled in XML in a similar manner.
3.4. MAC Protocol Designer

3.4.3 Code Loader

The Code Loader layer interacts directly with XML model layer to load the XML in RAM in the form of Java objects. These objects are populated with the state information of the nodes from XML and are used in the auto code generation process.

3.4.4 TinyOS 2.x Code Generation

The Code Generator layer parses and translates the Java objects into auto-generated TinyOS 2.x source code, which is then compiled and directly deployed onto the targeted platform. We have specially considered that the auto-generated code is organized so that advanced users may easily make modifications and build custom applications on top of it. We have implemented a number of MAC protocols using the framework including S-MAC [117], B-MAC [55], MFP-MAC [58] and X-MAC [96]. The implementations provide a well-defined MAC API to allow interaction of the MAC to the application and to control the MAC parameters. For instance, the interface of the MFP-MAC below contains the common functions to send/receive packets and to control the LPL operation of the MAC protocol.

```java
interface MFP_MAC {
    command void setCheckInterval(float interval);
    command void setDutyCycle(float dutyCycleValue);
    command uint16_t getCheckInterval();
    command float getDutyCycle();
    command void init(bool ENABLE_ACK);
    command error_t send(message_t* msg,uint8_t preambleType, uint8_t len, am_addr_t destAddress);
    event void sendDone(message_t* msg, error_t error);
    event message_t* receive(message_t* msg, uint8_t len); }
```
3.4.5 Performance Evaluation

In order to verify the validity and the functionality of the MAC protocols implemented using the MAC-PD, we have conducted goodput analysis for well-known MAC protocols. We have also carried out comparative empirical studies on these metrics for different MACs to their monolithic counterparts and those developed through MLA [148].

![Figure 3.20: Aggregate goodput of B-MAC developed using MAC-PD at different Packet Generation Rates (PGRs).](image)

In order to observe the performance characteristics of MAC protocols developed through MAC-PD, we replicated the experimental scenario for B-MAC as described in [55]. A receiver is placed in the centre and the number of equidistant transmitters (at a radius of 2 m) around it is gradually increased. All the nodes used a transmit power of 0 dBm. In order to calculate the achieved goodput, we measured the number of packets unicasted from each transmitter and correspondingly received. All the nodes used a fixed sampling period of 100 ms and a packet size of 32 bytes. The theoretical maximum throughput for these experiments is 2.415 kbps. Each experiment lasted for 120 s and is repeated three times. The mean aggregate goodput in Figure 3.20 shows identical trend as observed by the authors of B-MAC [55]. With a sampling period of 100 ms, the network is in saturation at low Packet Generation Rates (PGRs). Therefore, the aggregate goodput remains similar despite increasing the number of transmitting nodes. However, for traffic generation rates of
250 ms, 500 ms and 1000 ms, the network is not in saturation and therefore shows an increasing trend in the goodput with higher number of transmitters. These results indicate that MAC implementation using MAC-PD shows exactly the expected behaviour described in [55]. It also indicates that protocol performance is not hindered by componentizing the underlying MAC functionalities. Furthermore, we compared the goodput of B-MAC implemented using MAC-PD to the implementation using MLA with identical parameter settings. Figure 3.21 shows that both MLA and MAC-PD shows comparable performance characteristics and hence to monolithic implementations [114].

3.5 SUMMARY AND DISCUSSIONS

In this chapter, we have described the design details of a flexible MAC development framework based on the MAC decomposition philosophy. Our Decomposable MAC Framework enables high level of code re-usage, easy portability among different platforms and provide means for achieving rapid MAC protocol design and prototyping. We have described a list of the identified common MAC components and presented the interfaces to access these components. We expose granular blocks through flexible functional abstractions in order to provide a closer access to the PHY/MAC controlling parameters.
than contemporary closed MAC development frameworks. This certainly widens the experimental room for implementation of cross-layer designs. We have prototyped our framework on WARP boards with custom modifications of the hardware on FPGA for performance enhancements. We have implemented a large set of MAC protocols including IEEE 802.11 MAC like protocol with advanced features such as data aggregation and block acknowledgement. CogMAC, a spectrum agile MAC protocol which is suitable for CRNs, has also been implemented using our framework. The implementations themselves indicate that our framework enables rapid MAC prototyping.

Furthermore, we have implemented the same set of basic MAC components and different second level components which are targeted for more energy-aware sensor nodes for TinyOS 2.x. We observe a high level of component and code reuse for MAC realizations. Our framework imposes negligible memory consumption overhead on TelosB node as compared to the hand-coded monolithic implementations. Based on these components, we present MAC-PD, a protocol designer which allows rapid prototyping of MAC protocols with user friendly GUI. MAC protocols can be realized by “drag-drop-and-connect” components together. Prior knowledge of the platform and the language syntax is no longer required. MAC-PD generates a clean downloadable code for TinyOS 2.x operating system and currently supports TelosA, TelosB, MICAz, BSN-node and Intelmote2 nodes. Care has been taken that the auto-generated code is clean to allow the possibility of modification by advanced users. Experimental performance evaluation for various commonly used MAC protocols indicates that the MAC protocol generated by MAC-PD achieves comparable throughput as other reference MAC implementations.

The decomposable MAC framework serves as a foundation work in this dissertation. The component-based approach provides the opportunity for tool development to enable MAC reconfiguration with a high level of flexibility and a low level of implementation overhead. Decoupling of MAC functionalities also gives a new angle of investigating into the possibility of parallel execution of MAC processes. In the next chapter, we are presenting a toolchain for runtime MAC protocol realization which uses the decomposable MAC framework.
A Toolchain for Runtime MAC Protocol Realization

Over the past decade, wireless technology has gained much popularity due to its low cost, ease of use and wide range of end user services. Wireless communication is becoming an integral part of our daily life as development of applications and devices such as smart phones, building surveillance, and medical applications continues and expands. As a consequence, the demand for spectrum, interference mitigation, and symbiotic coexistence of different wireless devices is rising. Since wireless spectrum is a limited resource, an imprudent use would lead to its depletion and thereby hinder the growth of wireless services. Furthermore, as the range and nature of the applications for wireless networks increases, a particular wireless network is expected to support multitude of applications with varying requirements and different levels of QoS. A static and one-size-fits-all network solution is no longer to provide satisfying performance at all time. Moreover, in order to be able to fully utilize the spectrum in both ISM and licensed bands, a wireless terminal should have the ability to sense, learn and be aware of the parameters related to the radio channel, availability of spectrum and power, the operating environment of a radio, user requirements and applications, available network infrastructure, local policies and other operating restrictions as described in the context of cognitive radios [69,139]. Therefore, a MAC layer designed for these wireless terminals should be able to adapt based on the prior knowledge and subjected conditions so as to offer optimal performance characteristics. Since classically MAC protocols are designed to be highly optimized for a particular application in a static environment, a new approach is needed to provide the adaptability required.

Based on Decomposable MAC Framework which we have introduced in Chapter [3], we present a Toolchain for RUtiMe Protocol realization (TRUMP) in this chapter. TRUMP allows fast and flexible MAC protocol realization. Our approach enables wider experimental room and facilitates innovative MAC algorithm design for new generation of wireless networks. TRUMP enables both user and self-triggered reconfiguration strategies for protocol compositions at runtime. Our experimental results show that the configuration delay using our toolchain is in the order of a few microseconds on commercially available SDR platforms. This makes it possible for the protocol stack to meet the requirements for adaptability, flexibility and strict timelines of wireless networks.
To enable runtime protocol realization with minimum user-effort, an efficient language syntax for MAC protocol description and the associated compiler are designed. The language has a simple syntax but is expressive enough to realize and compose complex protocols as we will show later. It also offers the possibility of automating code generation and adaptation with cognition.

Our toolchain is lightweight, does not require heavy compilation during runtime and can easily be deployed on a wide range of commercially available platforms. We describe the system architecture of TRUMP in Section 4.1 and present the evaluation results on its WARP board implementation in Section 4.2. To address the resource constraints on embedded network platforms, we have adapted the toolchain design targeting wireless sensor nodes. The design of the toolchain is presented in Section 4.3 and the evaluation results described in Section 4.4. We conclude in Section 4.5. This chapter is mainly based on our articles [150, 151] which were published during the dissertation work.

4.1 TRUMP System Design

In this section, we describe the design details of our toolchain for realizing reconfigurable MAC protocols. As shown in the system architecture illustration in Figure 4.1, our toolchain is composed of three parts: a MAC development environment which mainly consists of a meta-language descriptor and can be extended with a graphical user interface; a MAC meta-compiler which translates a MAC description in meta-language into executable instructions for the target platform; a Wiring Engine which controls the execution of the designed MAC protocol. The design and implementation of this toolchain has been carried out using bottom-up approach as orderly described in this section. The toolchain works on a library of reusable MAC components as our Decomposable MAC Framework described in Chapter 3.

4.1.1 Component-based Framework

Performance of wireless networks in terms of throughput, latency, etc., is often unstable due to varying environmental conditions and changing applications. Stringent protocol designs tend to restrict adaptation of wireless networks according to the dynamic changes in applications, user behaviour and network conditions. Designing protocols in a modular way is therefore very promising as acknowledged by many researchers [29, 32, 97]. Our designed system works on component-based framework where both input and output of the components are clearly defined. As presented in Chapter 3, MAC protocols can be decomposed into elementary components such as Carrier Sensing, Timer, Radio Control, Backoff, Frequency Selection, etc. The framework provides fundamental building blocks for various protocol realizations. Although our current TRUMP implementation is based on Decomposable MAC Framework, in principle any component-based MAC framework can be supported where
components are reusable and well-defined input/output interfaces are provided.

4.1.2 Wiring Engine

In order to bind components together and execute the resulting protocol composition efficiently, we answer three major questions through TRUMP:

1. How all the components can be represented using unified interfaces.
2. How the logical relationship among components is expressed.
3. How a particular protocol is constructed and how the execution flow of components is maintained and altered.

Wiring Engine defines a unified API for all the functions and logical connections. After analyzing a list of data structures for possible representation of the composed protocol such as tree, graph, etc., we chose to use a linked list structure to store the composition of protocols due to its ease of management while satisfying our needs. The protocol is presented in terms of both functional and
logical components defined in an instruction set. A logic controller takes care of the execution flow of the protocol. Details of each component of the engine are described in the following paragraphs.

Component Interfaces

In our implementation, all the components have a standard interface format which takes a void pointer as parameter and returns an integer. For example, for the Send Packet component, `int SendPacket(void* para)` is used as the interface in the Wiring Engine. The interface allows an arbitrary number of parameters to be passed for all the components. Parameters can be stored in different data structures for different components but only the address of the data structure is passed as the input of the component. An example of API function definition is shown as below in C-programming style. The function `CarrierSensing()` takes two integers as parameters which are defined in the parameter list structure `cs_para`. In order to map this generic API to the specific API of the CS component, a type cast is performed to convert the void pointer to the type of the data structure we have defined for the CS function. The input parameters of the CS component can be then retrieved easily. The decomposable MAC APIs developed in Chapter 3 have been easily wrapped by these generic APIs.

```c
// API definition for
// int CarrierSensing(int csDuration, int csThreshold)

typedef struct _cs_para
{
    int csDuration; // Duration of CS
    int csThreshold; // Detection Threshold of CS
} cs_para;

int CarrierSensing(void* Para_List)
{
    cs_para* myPara = Para_List; // type cast
    // actions...
}
```

Linked List

We use linked lists to realize desired protocols by re-/linking the components together. Modification of a linked list is flexible and fast, which meets the requirement for speedy and dynamic adaptation of MAC protocols. Since all the components share a unified API, we can easily use a function pointer to address any component in the library. We use a function pointer list structure where functions are linked as elements in a list. The linked list supports operations such as dynamically adding/deleting/reordering of components. Since
MAC protocol behaviour is dictated by the components used and the execution order of them, runtime composition of protocols is easily realized by dynamically changing the linked list elements. A conventional array is also an option for sequential data storage; however, it is more difficult to manoeuvre for dynamic adaptation of array connections.

**Instruction Set**

Apart from the MAC functional components, arithmetic, binary and logical expressions are necessary in order to realize the complete state-machine of a MAC protocol. These expressions enable functions such as linking components base on conditions, declare and process parameters and variables, etc. In order to minimize implementation complexity and memory overhead as required by embedded platforms in general, we design a small instruction set including CONST, VAR, FUNC, IF, ELSE, END, LABEL, and GOTO for logical operations and expressions in realization of protocols. The instructions are represented as the elements in a linked list. Executing a MAC protocol in the form of linked list is done by traversing the whole list from head to tail. The structure of a node in the linked list is defined as shown in Figure 4.2. The

![Figure 4.2: The structure of a node in a linked list.](image)

Node ID is the identifier of the node. It specifies the position of the node in the list for inserting and deleting operations of the list. The data segment stores the actual information of the node, leading by one instruction from the instruction set. Two pointers are used for each node. One pointer points to the next node in the linked list and the other pointer points to node to be executed next based on the logic instructions. For example, for an IF instruction, the jump pointer points to the node of the matched ELSE clause in case the function returns FALSE.

**Logic Controller**

The logic controller acts as the brain of the toolchain. Since logical operations are included in the nodes, the sequence of program execution involves runtime
feedback and decision making. The logic controller decides on the next node to be executed on-the-fly based on i) the return value of the function, ii) the logical operations associated, iii) the dependencies among functions and iv) the availability of resources for execution. It ensures smooth execution of the composed protocol by identifying conflicts between the relationship among components and the execution order.

Dependency Management  In MAC protocols, some functional components are dependent on each other. In order to reduce the design efforts and offer designer the reassurance that only logically correct designed protocol can be synthesized and deployed onto the target platform, we have devised pre-defined rules to express interdependencies among components and incorporate them as part of the functional component libraries. There are two rules for the design decision:

- No dependencies exist in binding two objects together, i.e. if two functions are entirely dependent on each other, they should be defined as a single component.

- All dependencies are uni-directional, i.e. the declaration of objects $A$ depends on $B$ only shows restrictions of $A$ to $B$ but not vice versa.

There are three types of dependencies which are expressed in our system:

1. An unconditional dependency is usually related to hardware dependent functions. For example in a MAC protocol design, when a packet is to be sent, i.e. $f_a$ denotes SendPacket(), $f_b$ is WriteToTxBuffer(). We define $f_a$ is unconditionally dependent on $f_b$, i.e. ($f_a << f_b$) since the transmission buffer should always be filled with relevant data before actual transmission.

2. A conditional dependency is usually a user-defined common logical action. For example, function $f_a$, SendPacket() can be executed as it is but if $f_c$ CarrierSensing() is present, then SendPacket() is executed after CarrierSensing(), i.e. ($f_a < f_c$).

3. Furthermore, TRUMP allows an explicit indication of parallel dependency between functions for parallel execution. It is defined for functions with no conflict about hardware resources and logical relations. For example $f_b$ WriteToTxBuffer() does not share hardware resource with and can be executed at the same time as $f_d$ ReadFromRxBuffer() ($f_b \parallel f_d$).

Dependencies can be expressed at two places. One set of global dependencies is used to indicate the default relationship between components. These can be expressed separately from the MAC protocol description. In addition, the user can also specify dependencies in the MAC description code using keyword
The defined dependencies for each function are stored in a resource table. The resource table is queried by the compiler at design time to ensure the integrity of the program. It is also queried at runtime by the logic controller to schedule the functions to be processed in the correct order. In order to store the defined dependencies, we have created a resource table which contains both function definitions and dependency information as shown in Figure 4.3b. Dependencies are given among functions and represented in the form of two-bit numbers, i.e., 00 means unspecified, 01 represents \( f_a \ll f_b \), 10 maps to \( f_a < f_b \) while 11 indicates parallel dependencies. One resource table entry stores only one API function definition and a bitmap which indicates all the global/default dependency relationships with others in the table. Since each dependency is represented with 2 bits, a total of \( 2N \) bits are required in total to represent all the dependencies for \( N \) entries in the table. The \( i^{th} \) group of 2 bits in the bitmap indicates a dependency relationship with the \( i^{th} \) function in the table.

In addition to \( N \) entries for \( N \) functions, the resource table is appended when the compiler extracts user-defined dependencies while parsing the MAC meta-description code. As shown in the resource table in Figure 4.3b, the first 8 entries represent the default relationship as described in the global dependency description. The 9th entry with index 0xF013 represents the line \( \text{dep}(\text{SendPacket} < \text{ReadFromRxBuffer}) \). In general, \( \text{SendPacket()} \) is independent from \( \text{ReadFromRxBuffer()} \) and these two function can be executed in parallel. However, in the sample protocol, the packet that is to be sent is actually an acknowledgement packet for the received data packet. Therefore, necessary header fields for the ACK packet need to be filled with the information the DATA packet provides, e.g., the destination address, the sequence number of the packet, etc. Therefore, an exception is indicated here. The MSB of the 16-bit index field is 0 for global dependencies and 1 for user-defined dependencies. The 2nd to 4th MSBs are reserved bits for possible extension of the resource table in the future. Currently the values are zero for all three bits in all entries for all entries. The 12 LSBs for global dependencies corresponds to the index of the function in the resource table while for the user-defined dependencies it represents the position of the function in the linked list. In the example shown in Figure 4.3a), the \( \text{SendPacket()} \) function is in line 19 (0x013) since the dependency exception should be applied only to that function in that particular position.

Since TRUMP keeps track of dependencies among all the functions within the library, we explore our opportunity in parallel execution of functions and possible advantages it can bring. When more than one processing element exist, independent functions are scheduled onto different processing elements and executed simultaneously. As shown in Figure 4.3e), the first three functions \( \text{WriteToTxBuffer()}, \text{BackOff()} \) and \( \text{SelectChannel()} \) are inde-
a) A simple CSMA based MAC protocol described in MAC meta-language.

```
label Start;
WriteToTxBuffer (DATA);
label TryToSend;
BackOff();
SelectChannel();
SetFrequencyChannel();
if(CarrierSensing())
  SendPacket (DATA);
else
  if(WaitForPkt (ACK))
    ReadFromRxBuffer();
  goto Start;
endif
else
  if(WaitForPkt (DATA))
    ReadFromRxBuffer();
  WriteToTxBuffer (ACK);
  dep(SendPacket < ReadFromRxBuffer)
  SendPacket (ACK);
endif
```

b) A resource table with both global dependencies and user-defined dependencies.

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
<th>Dependency Bitmap</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>WriteToTxBuffer</td>
<td>00 11 11 00 00 11 00</td>
</tr>
<tr>
<td>0x0002</td>
<td>BackOff</td>
<td>11 00 11 00 00 11 00</td>
</tr>
<tr>
<td>0x0003</td>
<td>SelectChannel</td>
<td>11 11 00 00 00 11 00</td>
</tr>
<tr>
<td>0x0004</td>
<td>SetFrequencyChannel</td>
<td>00 00 10 00 00 00 11</td>
</tr>
<tr>
<td>0x0005</td>
<td>CarrierSensing</td>
<td>00 10 00 10 00 00 00</td>
</tr>
<tr>
<td>0x0006</td>
<td>SendPacket</td>
<td>01 00 00 10 00 11 00</td>
</tr>
<tr>
<td>0x0007</td>
<td>ReadFromRxBuffer</td>
<td>00 00 00 00 00 11 00</td>
</tr>
<tr>
<td>0x0008</td>
<td>WaitForPkt</td>
<td>00 00 00 00 00 00 00</td>
</tr>
<tr>
<td>0x8013</td>
<td>SendPacket</td>
<td>00 00 00 00 00 10 00</td>
</tr>
</tbody>
</table>

```
c) The compiler i) converts MAC description into a linked list and ii) extracts dependency from the meta code and completes the resource table.
```

d) Default dependencies for all MAC protocols.

```
WriteToTxBuffer || BackOff || SelectChannel || ReadFromRxBuffer ; CarrierSensing < BackOff ;
CarrierSensing < SetFrequencyChannel ;
SetFrequencyChannel < SelectChannel
SendPacket <<< WriteToTxBuffer
SendPacket < CarrierSensing
ReadFromRxBuffer || SendPacket ...
```

e) The logic controller governs the execution order of the components based on the dependencies in the resource table and the availability of processing entities.
dependent from each other and thus scheduled onto three processing elements PE1, PE2, PE3 for execution at the same time, respectively. SetFrequencyChannel() follows SelectChannel() immediately and CarrierSensing() can only be executed after both BackOff() and SetFrequencyChannel(). Some results on benefits brought by parallelization is evaluated in Chapter 6.

**Error Detector**  Error detection is another main functionality provided by the logic controller in component-based MAC protocol realizations and adaptations. When a MAC protocol is created or modified at runtime, we see potentially two types of errors. Syntax-related errors refers to problems such as unmatched IF-ELSE statements or lacking of LABEL for GOTO statements. The other type of errors is dependency conflict, e.g., the MAC protocol tries to send a packet when the radio is at OFF state. Therefore, the logic controller provides error detection at runtime against these two types of errors. The syntax error checking action is carried out whenever a new protocol is executed or a modification is made while the dependency error detection is carried out whenever a function is to be executed at runtime.

**Syntax Error Detection**  Syntax error detection is mainly responsible for identifying unmatched IF-ELSE-ENDIF and GOTO-LABEL pairs. For each GOTO statement, the Error Detector goes through the protocol list to find the matching LABEL. Error checking for IF-ELSE statement is comparatively more complicated and has been described in Algorithm 1. In the process of checking, a static stack is used to store all the IF, ELSE and ENDIF nodes present in the list. IF nodes are simply pushed onto the stack. When there is an incoming ELSE nodes, we first check first the top element of the stack. If the top element is an IF node then the ELSE node is pushed onto the stack, otherwise, an ERROR message is returned. Once an ENDIF node arrives, the top pair of IF and ELSE nodes are popped out. At the end of traversing the list, an empty stack indicates no syntax error is found in the protocol design.

**Dependency Error Detection**  The dependency error detection is achieved through the dependency bitmap in the resource table. A bitmap called FuncBitmap is used as a runtime function queue to record all the current running functions. When a function has cleared its dependency conflict and is put to execution, the bit mapped to the index of this active function is set to be 1. Similarly, when a function is done execution, the corresponding bit will be set to 0. When a function is the next in-line to be executed, a dependency check is performed. The resource table is queried and the relevant information in the resource table is retrieved based on the indication in the FuncBitmap. When no 10 or 01s are found, which indicates that the function in-waiting is not dependent on the completion of any of the currently executing functions, the function will be executed in parallel with the rest. Otherwise, the execution
Data: List
set stack to empty;
CurrentNode = List.head;
while CurrentNode ≠ List.tail do
    type = CurrentNode.type;
    switch type do
        case IF
            stack.push(IF);
        case ELSE
            if stack is empty then
                return IF_ERROR;
            else
                top = stack.pop();
                if top ≠ IF then
                    return ELSE_ERROR;
                end
            end
        case ENDIF
            if stack is empty then
                return IF_ERROR;
            else
                top = stack.pop();
                if top == ELSE then
                    stack.pop();
                end
            end
    endsw
    CurrentNode = CurrentNode.next;
end
if stack is empty then
    return NO_ERROR;
else
    return END_ERROR;
end
Algorithm 1: Syntax error detection algorithm to identify errors in IF-ELSE statements.
of the function will be delayed till the one function in the stack has been executed and the FuncBitmap has been updated. The dependency error check will be performed again till no dependency conflict is found.

4.1.3 Meta-language Descriptor

In order to facilitate efficient protocol implementation, we have defined a MAC meta-language and a corresponding compiler (cf. Section 4.1.4). We use a C-like language with keywords if, else, endif, label, goto corresponding to the instruction sets for Wiring Engine. Figure 4.3a) shows an example of a CSMA based MAC protocol described using the meta-language. The keyword dep is used to indicate dependencies among components in the MAC descriptor.

In order to facilitate the design process of MAC protocols for users who are not familiar with programming languages, following the approach of MAC-PD (cf. Section 3.4), we have also developed and provided a GUI for rapid MAC prototyping. Unlike MAC-PD, which is a full-blown toolchain that stores the MAC flowchart representation in XML and generates TinyOS code for target platforms, the GUI we have designed as part of TRUMP matches to our meta-language descriptor. It shares the same “drag-and-drop” concept in protocol visualization as MAC-PD and is independent of the target platforms. A designer can drop selected components in the design pane, set individual component parameters and connect the components in desired order. The meta-language code is generated based on the designed flowchart by the interface. The user interface also checks the integrity of the user design and reports design error, e.g. missing a connection between components, incomplete logic expressions, etc. Figure 4.4 shows a screenshot of the MAC design interface.

In addition to convert the flowchart to meta-language code, which is then passed onto a meta-compiler to be translated to linked list for execution on the target platform, the GUI also implements a command shell. The command shell allows communication between the designer and the target platform. The commands included in the command shell are listed in Table 4.1. These commands can be used by the designer at runtime for MAC protocol modification and reconfiguration. Debugging information can also be viewed through the shell.

We have conducted tutorials and a winter school workshop introducing TRUMP as a tool for fast MAC protocol prototyping [152,153]. All participants managed to grasp the concept and designed fairly complicated and functional protocols within two hours using either the C like meta-language descriptor or the flowchart based graphical user interface which shows the user-friendliness of the bi-directional approach of our tool. More materials on the tutorials can be found at [152], [153].
Figure 4.4: Screenshot of the MAC designing tool.

Table 4.1: Command definitions in the shell implementation.

<table>
<thead>
<tr>
<th>Commands</th>
<th>Action Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>help</td>
<td>print manual of all commands</td>
</tr>
<tr>
<td>new filename</td>
<td>compile a new protocol from filename</td>
</tr>
<tr>
<td>print</td>
<td>display the current protocol</td>
</tr>
<tr>
<td>add linenum</td>
<td>add a new line after linenum</td>
</tr>
<tr>
<td>delete linenum</td>
<td>delete the line number linenum</td>
</tr>
<tr>
<td>run</td>
<td>execute the protocol list</td>
</tr>
<tr>
<td>stop</td>
<td>stop the execution of the protocol list</td>
</tr>
<tr>
<td>check code</td>
<td>check for errors in the protocol list</td>
</tr>
<tr>
<td>prl code func1 func2 ...</td>
<td>enable concurrent execution of func1 func2 ...</td>
</tr>
</tbody>
</table>

4.1.4 Meta-compiler

A compiler is implemented as part of the toolchain to convert protocol description written in our meta-language protocol descriptor to linked list which realizes protocol implementation. The compiler is developed using Lex&Yacc [154]. The compiler consists of a scanner and a parser. While the scanner reads the code and scans tokens to the parser, the parser checks the syntax and builds the
output list. Following the appearance order, each single instruction defined in meta-language is compiled to one node and appended to the list. The variables and constant values are respectively added into variable and constant pool by the compiler and the memory addresses are mapped to all their references. The compiler also extracts dependencies among function as specified by the user in the MAC description code and inserts them to the resource table for the logic controller to govern the execution order of the protocol.

Our meta-compiler also facilitates the MAC designing process of the user by providing the predicted protocol performance at design stage. Prior knowledge of the execution speed and power consumption of all components are fed into the system. Estimated protocol performance is calculated based on the components used and the associated performance metrics. Therefore, the designer can optimize the design based on the predicted performance information without having to deploy the MAC code on the target platform. Of course the estimated protocol performance just provides a general guideline for designer since it can differ from the actual performance based on the operating environment, placements of the nodes, etc.

Furthermore, the meta-compiler functionality has been extended to facilitate MAC protocol optimization. The designer can specify preferences for the MAC protocol performance such as minimum energy consumption, minimum latency, maximum data rate, etc., through the meta-compiler. The compiler assists in selecting MAC protocol parameters, components which suits to the user preferences and conveys the preference for runtime protocol optimization as described in Chapter 5.

4.2 Evaluation Results of TRUMP on WARP Boards

To validate our design, we have implemented TRUMP on WARP SDR boards. WARP board v1 is a popular SDR development platform which offers a wide range of capabilities in using and modifying the radio and PHY layer functionalities for MAC protocol implementations. Our Decomposable MAC Framework has been realized on WARP for TRUMP implementation as described in Chapter 3. WARP board has a Virtex Pro II FPGA with PowerPC core synthesized and a default on-chip memory of 128 kB which is extendable to 256 kB. It has 4 MB off-chip memory which requires to be initialized using a bootloader. The radio daughter board consists of flash ADCs, a DAC, a dual-band power amplifier and MAXIM MAX2829 RF transceiver. In order to evaluate the performance of TRUMP, we focus mainly on two aspects: overhead in terms of memory usage and execution time. At the time of our experiments, no operating system is installed on WARP and it supports only sequential function execution. Therefore, we were not able to evaluate the advantages offered by parallel dependency management on WARP which we have envisioned as explained in Section 4.1. The potential in parallelization will be later evaluated.
on multi-core platforms to be presented in Chapter 6. We have implemented a number of MAC protocols using TRUMP on WARP boards for our experiments and evaluations. We have measured the memory footprint and protocol execution time using TRUMP in comparison to monolithic implementation without TRUMP. User interaction with WARP board is carried out through serial communication over UART. We have carefully removed the serial communication delays in the results presented.

4.2.1 Memory Usage

Table 4.2 shows the size of the executable ELF file for the PowerPC on the WARP board. .text represents the size of the text section, .data indicates the size of the Read-Write data section, the value of .bss is the size of the uninitialized data section and .dec is the total size of the program. As compared to the monolithic implementation of protocol, our TRUMP tools with Wiring Engine have generated approx. 20kB code size. TRUMP imparts about 10 KB code size (.text) which is considerably small and can be deployed in most of the commercially available embedded platforms. A significant increase in the program memory (.bss) usage due to the static memory model which is used for linked list structure since dynamic memory model works poorly on WARP without proper dynamic memory allocation support. The compiler generated by Lex & Yacc adds a significant overhead although it is still within the on-chip memory limit of WARP board. Please note that the code size displayed in Table 4.2 does not differ with different MAC protocol implementations. TRUMP allows on-the-fly realization and configuration of MAC protocols. All the components have been initialized so that they can be used at anytime during the course of protocol execution. Since we have chosen a static memory model for our implementation to achieve stable protocol performance, no extra memory is needed to realize a wide range of MAC protocols.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>.text</th>
<th>.data</th>
<th>.bss</th>
<th>.dec</th>
</tr>
</thead>
<tbody>
<tr>
<td>without TRUMP</td>
<td>122737</td>
<td>1640</td>
<td>17280</td>
<td>141650</td>
</tr>
<tr>
<td>with Wiring Engine</td>
<td>132405</td>
<td>1732</td>
<td>29320</td>
<td>163457</td>
</tr>
<tr>
<td>with compiler</td>
<td>191245</td>
<td>1636</td>
<td>37680</td>
<td>230761</td>
</tr>
</tbody>
</table>

4.2.2 Execution Time Overhead

Table 4.3 shows the execution time of three MAC protocol implementations. Simple ALOHA uses WaitForPkt(), WriteToTxBuffer() and SendPack
etPacket() while CSMA protocol further includes CarrierSensing(). The Spectrum-Agile MAC (CogMAC) protocol additionally uses SetChannelPool() and SelectChannel(). The MAC description code for these three MAC behaviours is shown in Figure 4.5. The code is colour coded to show the reuse of components when changing MAC behaviours from one to another. The packet transmission measured in this experiment for both the header and the payload is with BPSK modulation scheme. The lengths for header and payload are selected to be 24 bytes and 1000 bytes respectively. We show that with three different implementations, the overhead caused by using TRUMP for servicing the node, locating the function, etc., is in the order of microseconds and within the 1% bound in terms of execution time. In general, the overhead of traversing one list increases with the complexity of the list. It also depends on the position of the functions used in the component library.

We have also measured the re-configuration delay and execution time of various commands offered by the command shell that can be performed to a protocol at runtime. The results are presented in Figure 4.6 for different pro-

Table 4.3: Execution time of different MAC protocol implementations using Decomposable MAC Framework on WARP board with and without TRUMP.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>ALOHA [ms]</th>
<th>CSMA [ms]</th>
<th>CogMAC [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of components in the list</td>
<td>5</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>Execution time w/o TRUMP [ms]</td>
<td>1.491</td>
<td>1.503</td>
<td>1.537</td>
</tr>
<tr>
<td>Execution time with TRUMP [ms]</td>
<td>1.495</td>
<td>1.513</td>
<td>1.548</td>
</tr>
<tr>
<td>Execution time overhead [us]</td>
<td>4</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>
Figure 4.6: Execution time when different commands are executed for MAC protocols of different complexities.

tocol complexities. The commands are selected from Table 4.1 in Section 4.1. Add and Delete commands add and delete nodes in the protocol list. These are commonly used in modifying protocol realizations. We have taken the measurements where the add/delete operations are performed to the last node of the list, i.e. a node is append to the end of the list or the tail node is deleted. This is to show the worst case scenario since the pointer used to service the protocol always starts from the beginning of the list. The Check instruction performs error detection for both syntax errors such as missing logic operators and dependency conflicts. Error messages are returned to help the MAC protocol designer to modify the protocol description. Print is a debug friendly command which displays the content of the current protocol list. Since all the command requires servicing the entire linked list, the execution time is dependent on the size of the list. We see that the time increases linearly with the number of nodes in the scale of microseconds. Please note that we have removed the serial communication delay from the user end to the target platform. Since these commands can be used to enable automatic protocol reconfiguration without user inputs at runtime, the measurements for especially the Add and Delete commands also indicate the protocol reconfiguration time using TRUMP.
Wireless sensor networks are being deployed in a wide variety of applications, especially in embedded networking domain. Data reliability, latency tolerance, volume of generated traffic, and expected life-times vary hugely from one application to another. Hence, there is a need for different types of sensor network MAC protocols supporting different applications and deployment conditions. Developing new MAC protocols is, however, a cumbersome task and typically requires domain expertise and platform specific programming skills. Moreover, the time and effort required for testing and debugging newly developed code on embedded sensor nodes are high. Most of the existing protocols are developed with a monolithic programming style, which restricts code reusability and portability for multiple platforms.

In MAC-PD [125] (cf. Section 3.4) as the first step to enable flexible MAC protocol realization for WSNs, we have used basic functional components (cf. Section 3.2) to form existing protocols as well as non-classical MAC protocol designs. In addition, MAC-PD enables easy MAC design through a drag & drop based graphical user interface [126]. However, this tool did not target runtime reconfiguration options without needing to reprogram the nodes. Additionally, although the philosophy of designing MAC protocols in the form of flowchart can be favourable for certain groups of designers, we have learnt in different workshops/tutorials [152, 153] that some designers prefer ‘code-like’ syntax.

In this chapter, we have introduced TRUMP, a toolchain which offers the capability for runtime protocol realization and configuration. In this section, following a similar design philosophy, we have extended the toolchain for resource constrained sensor node platforms in order to provide both easy MAC design and runtime reconfiguration capability. We present a set of tools designed to address the resource constraints of sensor networks while allowing fast prototyping and runtime reconfiguration. We compare our implementation to the hand coded monolithic implementation for different well-known MAC protocols on commercially available sensor nodes. We show that while enabling flexibility and reconfigurability, the execution time overhead is less than 1.4% for our measured worst case. The time required for protocol reconfiguration has been reduced significantly by up to 60% compared to the conventional reprogramming approach. The ability to redirect, add, delete and modify components in the state machine on-the-fly allows runtime protocol reconfiguration of a sensor network to be easily realized by over-the-air programming.

As our effort to extend TRUMP into resource constrained platforms, such as sensor nodes, we have designed the overall architecture of our MAC toolchain as shown in Figure 4.7. Due to the severe resource constraints, we have
offloaded part of the toolchain from the target platform to the more capable PC while providing similar development experience. As shown in Figure 4.7, the MAC protocol description written in user-friendly meta-language is translated to a sensor node interpretable format through a meta-language parser into an array structure. The array is referred to as MetaItem list. The MetaItem list which encompasses the entire MAC description is then passed to sensor nodes through serial communication and executed on the target platform by a MetaItem executor. The MetaItem executor also manages the list of variables used in the protocol and maps MAC components in the component library to the MetaItem for execution at runtime. The system architecture is organized in a way that although a workstation (e.g. PC) is needed for prototyping and debugging of newly designed MAC protocols, sensor nodes operates autonomously for protocol execution and modification. Therefore, runtime protocol reconfiguration of a sensor network can be easily realized by over-the-air pro-
4.3. RECONFIGURABLE MAC PROTOCOLS FOR RESOURCE CONSTRAINED PLATFORMS

gramming, i.e. reconfiguration commands can be encapsulated with standard message format to be propagated through the network and the MetaItem executor modifies the MetaItem list by interpreting the message.

Since TinyOS [147] is a commonly used embedded operating system targeting WSNs and supports a wide range of sensor node platforms such as TelosB, MICA2 and MICAz from MEMSIC [123], etc., we have targeted the toolchain for TinyOS. In order to show the portability of our toolchain among various sensor node platforms, we have chosen TelosB and MICA2 as our target platforms since these two commonly used platforms have different radio chips and microcontrollers. In the rest of the section, we describe the design and implementation details of each part of the toolchain in a bottom-up approach to show how a fully flexible and reconfigurable MAC protocol is realized. We have engineered the toolchain to overcome the limitations imposed by TinyOS such as lack of support for dynamic memory allocation.

4.3.1 MAC Component Library

Our tool is built based on Decomposable MAC Framework [124]. As described in Section 3.3, we have already implemented a list of components in TinyOS 2.x including Send Frame, Receive Frame, Radio Control, Timer, Random Number Generator, Carrier Sensing, LplCoordinator, Binary Exponential Backoff, Send Packet, Expect Packet and Send Preamble which are basic building blocks for a wide range of preamble-sampling based and common active period based WSN MAC schemes. In addition to our previous implementations for CC2420 radio based nodes, we have ported some of the basic level components using the radio driver functionalities such as Carrier Sensing and Send Frame for CC1000 radio chips (for supporting MICA2 nodes). The overall porting effort was little thanks to our generic component APIs. All the above identified component interfaces are kept unified and platform independent.

4.3.2 Execution Scheduler

Apart from the MAC functional components, arithmetic, binary and logical expressions are necessary in order to realize the complete state-machine of a MAC protocol. These expressions enable functions such as linking components base on conditions, declare and process parameters and variables, etc. The Execution Scheduler governs the execution of the MAC protocol by interpreting the expressions to form a correct execution sequence of the MAC components. As shown in Figure 4.7, there are three main components in the Execution Scheduler: a MetaItem list which stores the MAC protocol design in the form of a list of elements called MetaItem; a variable list which stores all the variables declared in the MAC design and a MetaItem executor which translates the MetaItem list into executions and manages the variable list accordingly.
MetaItem List

Our toolchain uses a linked list structure to describe a reconfigurable MAC protocol. As it is complicated to manage string variables in TinyOS, numerical values are used in the MetaItem structure as shown in Figure 4.8. The MetaLabel specifies the type of the MetaItem, for example DEF for a variable definition, FUNC for a function call, etc. IdLabel specifies the identifier of the node, for example the name of the function for a function call while Parameters are a set of attributes which the MetaItem requires.

```c
typedef struct MetaItem {
    uint8_t MetaLabel;
    uint8_t IdLabel;
    MetaParam Parameters[5];
}MetaItem;
typedef struct MetaParam{
    uint8_t type;
    uint16_t value;
}MetaParam;
```

Figure 4.8: Definition for MetaItem structure.

Variable List

There are three types of variables included in an array structure of pre-defined size of 50 elements:

- **System variables**: These are pre-defined by the toolchain to be used globally but they cannot be modified or re-defined at runtime.

- **Global variables**: These are defined by the user to be used within the scope of the entire protocol implementation.

- **Local variables**: These are defined by the user within an event implementation and are only valid within a specific event. These variables are removed from the list when the event execution is completed.

MetaItem Executor

The MetaItem executor governs the execution sequence of the MAC procedures by managing an execution pointer. The executor first analyzes the MetaLabel in the MetaItem to determine how to interpret the rest of the fields. It adds new variables to the variable list and removes invalid ones. It also translates MetaItems into function calls which are mapped to the MAC component library. The conditional branches such as IF-ELSE-ENDIF are interpreted
and the execution pointer is moved to the appropriate branch according to the conditions specified. The MetaItem executor also points to implementation of the event when it is triggered. An event is realized by defining two specific MetaItems, EVENT_NAME which contains the name of the event and END_EVENT, which specify the beginning and the end of the event, respectively. Last but not the least, the executor accepts the command for MAC protocol modification at runtime, e.g. to add/remove a MetaItem from the list and adapts the execution accordingly.

4.3.3 Meta-language Descriptions and the Parser

In order to provide a rapid MAC prototyping tool without requiring specific knowledge of the target platforms from MAC designers, we have introduced a C-like MAC meta-language for MAC design in Section 4.1.3. The language contains logical expressions such as IF-ELSE-ENDIF and LABEL-GOTO conditional clauses. The use of MAC meta-language simplifies the MAC design process. Several modifications have been made to the existing language architecture to suit to the characteristics of sensor nodes. The concept of event is introduced in the language. Since all the callback functions are predefined at compile time, the events associated with function calls are fixed, thus the names of all the events are pre-defined. Each of the event is mapped to a keyword string to be used in the meta-language. BOOT_EVENT is a default event that is triggered when the sensor node is booted. When the sensor node is booted, the MetaItem executor will move the execution pointer to BOOT_EVENT. Other events are associated with the function calls used in the implementation. For example, in the protocol description in Figure 4.9, startRadio() function is called in the boot event, therefore RADIOSTART_EVENT should accordingly be implemented.

In Figure 4.9, the translation procedure from MAC meta-language protocol description to the MetaItem list is illustrated. This is done through a parser implemented in Java. Each line of the protocol is parsed to a MetaItem structure which is fully compatible with the TinyOS structure defined in Figure 4.8. MIG (Message Interface Generator) tool is used to provide the structure of the compatible Java MetaItem object. The parser also verifies the meta-language syntax and reports errors, such as variables used before being declared.

4.3.4 Command Shell

We provide a similar set of commands as described in Section 4.1.3 for user interaction with the target platform and for runtime protocol reconfigurations. The command shell provides an interface for the MAC designer to a) load a new MAC design, b) execute the loaded MAC design, c) stop the current MAC execution, d) add new contents to the currently running MAC protocol, e) remove contents from the currently running MAC protocols and f) get
Figure 4.9: A mapping of protocol description in MAC meta-language to the MetaItem list for execution.

helpful information on how to use the command shell. The structure of all the commands are listed in Figure 4.10.

>> LOAD [filename]
>> RUN
>> STOP
>> ADD [numLine] [func/exp/def/if-else/label&goto]
>> REMOVE [numLine]
>> HELP [load/add/remove/run/stop/syntax/function/variables]

Figure 4.10: Commands defined for the command shell.

4.4 Evaluation Results on Sensor Nodes

In this section, we evaluate our toolchain in terms of memory consumption and execution time as compared to their monolithic counterparts as the clas-
4.4. Evaluation Results on Sensor Nodes

A practical way for protocol implementation. Since our toolchain introduces extra components to realize runtime reconfigurable solutions, we expect an overhead in terms of memory consumption. In addition, we have also introduced extra layer of abstraction in terms of MAC component implementation which can lead to an execution time overhead. Therefore, these two metrics are the trade-offs we made to enable reconfigurability, portability and code reusability. All the experiments were carried out on TelosB and MICA2 nodes with Texas Instruments CC2420 radio chip [155] and CC1000 radio chip [156], respectively. Please note that the monolithic implementation of all the MAC protocols in our evaluation is still based on the MAC components we have introduced in Section 3.2 but without all other parts of the toolchain. A selection of well-known preamble sampling based MAC protocols (i.e. B-MAC [55], MFP-MAC [35] and X-MAC [36]) and common active period MAC protocols (i.e. S-MAC [177] and T-MAC [41]) are implemented for evaluation.

4.4.1 Memory Consumption

Since our toolchain is tailored for resource constrained embedded platforms, memory consumption is an important metric. We have measured both RAM and ROM in terms of memory consumption. Table 4.4 shows the memory footprint results in terms of RAM and ROM consumption for a TelosB node for a list of different types of MAC protocols. No-protocol represents a simple procedure which only starts the radio, sends one packet, stops the radio, and repeats this process. We can see that the memory consumption for the monolithic implementations is based on the components used and varies among protocols. For the toolchain implementation, since all the components can be wired at runtime, the ROM consumption remains the same for all protocols. The RAM consumption is dependent on the protocol complexity, i.e. how many MetaItems are required to describe the protocol. The number of MetaItems for each protocol realization is presented in Section 4.4.3. A typical 2kB RAM overhead is observed for our implementations. In practise, one can maximize the usage

<table>
<thead>
<tr>
<th>Protocols</th>
<th>Monolithic</th>
<th>Toolchain</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ROM</td>
<td>RAM</td>
</tr>
<tr>
<td>B-MAC</td>
<td>25112</td>
<td>1242</td>
</tr>
<tr>
<td>MFP-MAC</td>
<td>25734</td>
<td>1242</td>
</tr>
<tr>
<td>X-MAC</td>
<td>25304</td>
<td>1242</td>
</tr>
<tr>
<td>S-MAC</td>
<td>24326</td>
<td>1306</td>
</tr>
<tr>
<td>T-MAC</td>
<td>22096</td>
<td>1140</td>
</tr>
<tr>
<td>No-Protocol</td>
<td>16978</td>
<td>1102</td>
</tr>
</tbody>
</table>

Table 4.4: Memory footprints [byte] of the implementations for different MAC protocols on TelosB nodes.
4. A Toolchain for Runtime MAC Protocol Realization

of the memory by declaring a maximum array size for the target platforms to host more complex MAC protocol realizations. It is worth noting that although the toolchain appears to impart a heavy memory consumption overhead, it allows realizations of all MAC protocols and switching among MAC behaviours at runtime without using any extra memory.

4.4.2 Execution Time

In order to evaluate how much the execution overhead the toolchain has introduced, we have measured the time taken to send a packet with varying payload with and without the toolchain on sensor nodes as shown in Figure 4.11. We have repeated each experiment 10 times to have better statistics and the average is presented. We have observed as expected a linear relationship between the packet sending duration and the packet size as shown in Figure 4.11. The overhead of using the toolchain is constant and almost negligible for both platforms. The time taken to send a packet of the same size is much longer on MICA2 than on TelosB due to the lower data rate supported by CC1000 radio as compared to CC2420 radio.

We have also measured the execution time of selected protocols on both TelosB and MICA2 nodes as shown in Table 4.5. As shown in Figure 4.11, since the absolute overhead is constant, the larger the payload size is, the less significant the overhead becomes. We have chosen a small payload size (11
4.4. Evaluation Results on Sensor Nodes

Table 4.5: Execution time [millisecond] for different protocols on TelosB and MICA2 nodes.

<table>
<thead>
<tr>
<th>MAC Protocol</th>
<th>Monolithic</th>
<th>Toolchain</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-MAC (TelosB)</td>
<td>161.78</td>
<td>163.36</td>
<td>0.97 %</td>
</tr>
<tr>
<td>S-MAC (TelosB)</td>
<td>195.84</td>
<td>198.57</td>
<td>1.39 %</td>
</tr>
<tr>
<td>No-protocol (MICA2)</td>
<td>26.67</td>
<td>26.88</td>
<td>0.79 %</td>
</tr>
<tr>
<td>B-MAC (MICA2)</td>
<td>190.49</td>
<td>191.13</td>
<td>0.33 %</td>
</tr>
</tbody>
</table>

bytes) for the rest of the experiments to show the worst case scenario for our toolchain. We can see that the overhead percentage added by the toolchain is around 1% in most of the cases on both platforms.

4.4.3 Reconfiguration Cost

In this section, we evaluate the cost to configure the MAC protocol at runtime. In the traditional monolithic approach, reconfiguration can be realized only by reprogramming the nodes. The measurements for reconfiguration shown in this section are taken from the user end between the time when a command for reconfiguration is issued and the response received from the target sensor nodes that the protocol has been transformed accordingly. Figure 4.12 shows the comparison for the time to load a completely new protocol between the monolithic implementation and reconfigurable approach using toolchain. For the toolchain approach, a new text file is loaded. We can see that in most of the cases 60% of the execution time has been saved.

We have also measured the time taken for the protocol to evolve base on reconfiguration commands. In our experiments, we send commands to change the duty cycle of the preamble-sampling protocols and the wake-up interval for the common period protocols. Table 4.6 shows that the reconfiguration time is independent from the complexity of the protocol and the reconfiguration response time for the user is about 50 ms. Since the time measured in-

<table>
<thead>
<tr>
<th>Protocols</th>
<th>No. of Metaitems</th>
<th>TelosB</th>
<th>MICA2</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-MAC</td>
<td>112</td>
<td>54</td>
<td>45</td>
</tr>
<tr>
<td>MFP-MAC</td>
<td>122</td>
<td>48</td>
<td>44</td>
</tr>
<tr>
<td>X-MAC</td>
<td>125</td>
<td>41</td>
<td>47</td>
</tr>
<tr>
<td>S-MAC</td>
<td>218</td>
<td>50</td>
<td>-</td>
</tr>
<tr>
<td>T-MAC</td>
<td>194</td>
<td>54</td>
<td>-</td>
</tr>
<tr>
<td>No-Protocol</td>
<td>12</td>
<td>31</td>
<td>47</td>
</tr>
</tbody>
</table>

Table 4.6: Time [ms] used to reconfigure a MAC protocol on TelosB and MICA2 nodes using our toolchain.
4. A TOOLCHAIN FOR RUNTIME MAC PROTOCOL REALIZATION

Figure 4.12: Comparison of the reconfiguration time on TelosB and MICA2 nodes.

volves multiple round-trip communication delay through serial port, we can deduce that the actual reconfiguration time on the target platform is much less.

4.5 SUMMARY AND DISCUSSIONS

MAC schemes require a high degree of flexibility and runtime adaptability in order to cope with the emerging challenges such as to satisfy varying application demands, user patterns, changing network conditions and efficiently utilize the spectral resources. In this chapter, we have described a toolchain, TRUMP, which allows runtime MAC protocol compositions by binding pre-defined reusable protocol components together. Our design provides fast reconfiguration speed and a lightweight implementation in order to meet time-critical requirements imposed by MAC processing. TRUMP provides a rich set of tools for developing complex MAC schemes including a MAC metalanguage descriptor, a MAC meta-compiler and Wiring Engine. A “drag-and-drop” based GUI is also provided to enhance user experience using flowcharts for designing MAC protocols. The implementation of TRUMP is done on the commercially available WARP boards using Decomposable MAC Framework as described in Chapter 3. TRUMP can in principle be used on top of any
component-based MAC framework where generic APIs are provided. Our evaluation results on WARP show that TRUMP enables fast runtime composition of protocols ranging from a simple CSMA based MAC to a multichannel spectrum agile MAC protocol with fairly advanced channel selection procedure [96].

Furthermore, we have adapted the design of our TRUMP toolchain to typically resource constrained platforms such as sensor nodes. We have implemented the toolchain on commercially available and widely used sensor nodes. We have conducted empirical studies to observe the overhead imparted by our toolchain. The results show that our implementation well fits the memory limit of the sensor nodes and execution time overhead using the toolchain is only 1% for the widely used preamble sampling and common active period MAC protocols. Although we have observed a typical 2 kB overhead in RAM for one MAC protocol implementation, our approach allows realizations of a vast list of MAC protocols at runtime using the given limited memory resource. Furthermore, as the demand for large-size low-power RAM increases largely due to the development of smartphones, we believe the availability of RAM should not be a bottleneck in allowing easy and rapid prototyping of MAC protocols. We have also observed that much smaller effort is required to develop and prototype new protocols using our toolchain compared to classical approaches. A demonstration video of our toolchain is available online [157].

In highlight, TRUMP provides the capability of MAC protocol performance optimization by offering easy reconfiguration and modification of MAC protocols. Since a wide range of protocols can be realized at runtime, TRUMP allows the current protocol to be adapted according to any changes in spectral environments, user preferences, and application requirements. TRUMP also offers dependency management among MAC components, which in turn provides the possibility to execute two or more independent components in parallel. When hardware resource is available, parallel execution of MAC components can bring significant performance improvement in terms of network latency and throughput. Based on the above mentioned two characteristics, TRUMP has enabled at least two major experimental rooms for MAC protocol research: MAC protocol optimization and parallel MAC protocol execution. We have investigated in these two areas and we are presenting the results in Chapter Chapter 5 and Chapter 6, respectively.
5

MAC Protocol Optimization

Static MAC protocols can be highly optimized for particular application scenarios in a stable spectral environment. However, wireless networks are unstable in nature: wireless link quality, i.e. the bandwidth available to the users, varies significantly over time, leading to unpredictable packet loss and varying level of QoS \[158\]; nodes join and leave a network randomly due to mobility or harsh environmental conditions \[159\]. Statically configured MAC protocols cannot cope with these dynamics \[160\]. Therefore, MAC protocols should be able to adapt and optimize at runtime in order to perform efficiently. Furthermore, the need for constant protocol optimization is rising also due to the emerging type of networks such as CRNs and active networks. CRNs advocates the ability of wireless terminals to sense, learn and be aware of the operating environment, application requirements and resource availability in order to fully utilize the spectrum \[69, 149\]. A MAC layer designed for these wireless terminals should be able to adapt based on the prior knowledge and subject conditions so as to offer optimal performance characteristics. As the ISM spectrum is becoming increasingly populated by emerging wireless networks, co-existence between heterogeneous networks has imposed challenges on the MAC layer operation. Adaptive MAC methods are necessary for the nodes to react to different network characteristics at runtime \[161\]. In this chapter, we present our solution to the following challenges in the realization of MAC optimization:

- A best-fit solution at all time is naturally highly desired. Typically, MAC protocols are designed for the application needs based on a rough estimate of the expected network topology and traffic load. This can yield a performance far off the application requirements \[162\]. In the past, there are two main approaches to reconfigure MAC protocol design to suit to changing application requirements and operating environments. One way is to choose a certain protocol or family of protocols for the application at hand \[163\]. For example, there is a vast list of hybrid MAC protocols such as Z-MAC \[54\] and Funneling-MAC \[56\] where the advantages of TDMA and CSMA based MAC protocols are used based on different traffic load and application scenarios. Meta-MAC protocol \[14\] is another example where any set of existing MAC protocols can be combined into a single upper layer to form the most suitable MAC solution at that time. The other way is to adjust the MAC parameters at runtime.
The most well-known runtime parameter adjustment is binary exponential backoff. Parameters such as transmission power, receiver sensitivities, modulation schemes, duty cycles, etc. are also considered often in contemporary flexible MAC designing. We found that the above mentioned two approaches are limited either by the range of pre-defined MAC protocols or the exposed tunable MAC parameters. Runtime MAC optimization should not only be achieved through traditional cross-layer design by appropriate parameter settings among different network layers; the ability to change the composition of protocol stack and thus modify the actual protocol behaviour is of high interest. In Chapter 4, we have introduced TRUMP, which enables runtime protocol realizations of a wide range of MAC behaviours. The MAC protocol realization uses Decomposable MAC Framework described in Chapter 3. Therefore, more efficient and effective runtime MAC protocol optimization can be realized using TRUMP.

• MAC protocol performance can cover a wide range of metrics, such as throughput, goodput, data robustness, latency, etc. Performance optimization requires specification of an optimization goal since some metrics are contradictory to each other and optimizing all metrics at the same time is impossible. Furthermore, the optimization goal based on changing application requirements can vary over time and space depending on the nature of the application and the events. Therefore, the MAC layer should be aware of the most updated criteria for optimization. For example, for a surveillance application using WSNs, minimum energy consumption is often the top priority for maximizing network life-time until when a critical events occurs and low latency overrides energy saving in order to deliver the information in the shortest time possible. There has been works on MAC parameter tuning towards single optimization goal such as energy consumption [164,165]. In our work, we have used a compiler-assisted approach to allow the application to indicate the optimization preferences at runtime. Metrics such as energy consumption, data reliability, network throughput can be selected by the application. MAC adaptation based on the optimizer is achieved using TRUMP through both parameter adaptation and MAC protocol re-composition.

• In a network, the nodes might experience different spectral condition or receive different QoS requirement due to the difference in terms of their physical location and the nature of the application tasks they are carrying out. Therefore, MAC optimization at individual nodes may lead to possibly incoherent MAC schemes in a neighbourhood. Incoherent MAC protocols can lead to disruption in communication links and thus a significant network performance degradation as a result of optimization. In this chapter, we propose a cooperative mechanism which is integra-
5.1 Optimization Problem

MAC protocol performance optimization is typically a multi-objective optimization problem. In this chapter, we consider three main performance metrics for optimization: energy consumption \( E \), throughput \( S \), and robustness of data delivery \( D \). The general definition of this multi-object optimization problem is as follows [169]:

\[
\begin{align*}
\text{Maximize/Minimize } & \mathbf{F}(\mathbf{x}) = [E(\mathbf{x}), S(\mathbf{x}), D(\mathbf{x})]^T \\
\text{subject to } & g_j(\mathbf{x}) \leq 0, j = 1, 2, ..., m, \\
& h_l(\mathbf{x}) = 0, l = 1, 2, ..., e,
\end{align*}
\]

where \( m \) is the number of inequality constraints, and \( e \) is the number of equality constraints. \( \mathbf{x} \) is a vector of design variables, in our cases a collection of MAC parameters and compositions. In the following discussions and experimentation of MAC optimizations in this chapter, we have selected different objectives and multi-object optimization methods. The mathematical representations of the multi-object optimization functions in this chapter are adapted from [169].

5.2 Runtime Optimization

Runtime optimization of MAC protocol is desired for a wireless network to adapt to its changing spectral environment and requirements promptly. In Chapter 4, we have introduced TRUMP, which allows composing a wide range of MAC protocols for various types of wireless networks at runtime by simply binding reusable MAC functional components. TRUMP enables flexible adaptation of a MAC behaviour on-the-fly. In this section, we describe a runtime
In order to optimize the performance at the MAC layer, we have identified three categories of parameters which need to be constantly monitored at runtime. First, the current operating network environment such as the spectral conditions in terms of either RSSI values or link quality indicator should be monitored. Observing the spectral condition gives the MAC layer a good idea of the channel condition and bandwidth availability which can be highly dynamic due to mobility in the network and coexistence with other networks in the same spectrum. Second, the application requirement parameters which dictate the priorities given to different performance metrics should be constantly updated. The application requirement has significant influence on the level of QoS the MAC layer is required to deliver. Last but not the least, the actual obtained performance characteristics should be fed-back to each node for online evaluation and adaptation accordingly. In this section, we focus on network environment monitoring and performance feedback monitoring while the application requirement monitoring is presented in Section 5.3.

Many cognitive radio and dynamic spectrum access algorithms leverage from machine learning algorithms for runtime optimization and adaptation of protocol behaviour according to dynamic spectral environment. These learning algorithms such as Genetic Algorithm, Bayesian reasoning, etc. can be used as the runtime optimizer for TRUMP since the parameters required for learning algorithms to form cost functions are easily extractable in our system and self-triggered reconfiguration is realized by modifying the wiring of the components. In our design, we have implemented a runtime optimizer which makes decisions based on a priori knowledge. We have constructed the runtime optimizer based on real performance measurements performed on WARP boards. Parameters including packet delivery ratio, throughput, current consumption, modulation scheme and interference level are used in the optimizer. We have carried out experiments to verify the effect of runtime optimization using both parameter adaptation and MAC behaviour adaptation using the same experimental setup as to construct the optimizer. Our experimental results indicate that through modifications of protocol compositions and MAC parameter tuning self-optimization of MAC performance is achieved. The integrity of the program logic and data flow are guarded by our toolchain that no erroneous protocol realizations are executed.

5.2.1 Experimental Setup

In order to construct the runtime optimizer and evaluate the effect of runtime MAC optimization enabled by TRUMP, we have established our experimental setup as shown in Figure 5.1. We use an Agilent E4438C vector signal generator to generate various interference patterns while an Agilent E4440A spectrum analyzer is used to verify the on-going transmission in the spectrum. An
5.2. Runtime Optimization

Figure 5.1: Experimental Setup.

Agilent Infinium DSO8104A oscilloscope with a sampling rate of 10 M/s is connect to an Agilent N2783A current probe to measure the execution duration of various MAC processes. The high level of sampling rate of the oscilloscope enabled us to have accurate timing measurements to the 10th of a microsecond. We have used two v1 WARP boards to form a transmitter receiver pair for all the experiments in this section.

5.2.2 Evaluation Results

Since TRUMP has enabled runtime MAC protocol adaptation by re-compose MAC procedures in addition to the traditional adaptation approach of parameter tunings, we evaluate TRUMP with our runtime optimizer using both the parameter adaptation based method and MAC re-construction on-the-fly approach. The implementation of the optimizer and TRUMP for all the evaluation results in this section is based on ODFM reference design v14 on WARP boards. We describe in detail the formulation of our runtime optimizer and the performance gain achieved. Prior comprehensive knowledge of all the components in the function library is essential to the optimizer to make decisions. We have measured the execution time for fundamental MAC processes on WARP as shown in Table 5.1.

The current consumption of WARP board v1 with one daughter board attached at different operating stages is measured using the current probe along the main power supply cable. The readings are listed in Table 5.2. The base current in supporting WARP board without any radio activity is 526.5 mA.
Table 5.1: Execution duration of MAC functions on WARP board.

<table>
<thead>
<tr>
<th>Function</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RadioToSleep()</td>
<td>3 $\mu$s</td>
</tr>
<tr>
<td>RadioToTx()</td>
<td>4 $\mu$s</td>
</tr>
<tr>
<td>RadioToRx()</td>
<td>4 $\mu$s</td>
</tr>
<tr>
<td>RadioTurnaround()</td>
<td>4 $\mu$s</td>
</tr>
<tr>
<td>SetFrequencyChannel()</td>
<td>22 $\mu$s</td>
</tr>
<tr>
<td>ReadRssi()</td>
<td>1.4 $\mu$s</td>
</tr>
<tr>
<td>CarrierSensing(1 sample,1 $\mu$s/sample)</td>
<td>3.7 $\mu$s</td>
</tr>
<tr>
<td>WriteToTxBuffer(1000 Bytes)</td>
<td>19.5 $\mu$s</td>
</tr>
<tr>
<td>ReadFromRxBuffer(1000 Bytes)</td>
<td>33.5 $\mu$s</td>
</tr>
<tr>
<td>CopyAmongRadioBuffers(1000 Bytes)</td>
<td>50.8 $\mu$s</td>
</tr>
<tr>
<td>CopyAmongPPCBuffers(1000 Bytes)</td>
<td>14.5 $\mu$s</td>
</tr>
<tr>
<td>TxPacket(1000 Bytes, BPSK)</td>
<td>1435 $\mu$s</td>
</tr>
<tr>
<td>TxPacket(1000 Bytes, QPSK)</td>
<td>758 $\mu$s</td>
</tr>
<tr>
<td>TxPacket(1000 Bytes, QAM16)</td>
<td>425 $\mu$s</td>
</tr>
</tbody>
</table>

Table 5.2: Current consumption for different operations on WARP board.

<table>
<thead>
<tr>
<th>Radio Status</th>
<th>Peripheral Servicing</th>
<th>Current</th>
</tr>
</thead>
<tbody>
<tr>
<td>Off</td>
<td>None</td>
<td>821.3 mA</td>
</tr>
<tr>
<td>Off</td>
<td>Constant polling</td>
<td>838.5 mA</td>
</tr>
<tr>
<td>Rx</td>
<td>Constant polling</td>
<td>861.8 mA</td>
</tr>
<tr>
<td>Tx</td>
<td>Constant polling</td>
<td>932.1 mA</td>
</tr>
<tr>
<td>Tx packet (0 byte) at 100 pkt/s</td>
<td>Constant polling</td>
<td>865.6 mA</td>
</tr>
<tr>
<td>Tx packet (1000 byte BPSK) at 100 pkt/s</td>
<td>Constant polling</td>
<td>891.7 mA</td>
</tr>
<tr>
<td>Tx packet (1000 byte QPSK) at 100 pkt/s</td>
<td>Constant polling</td>
<td>887.7 mA</td>
</tr>
<tr>
<td>Tx packet (1000 byte 16QAM) at 100 pkt/s</td>
<td>Constant polling</td>
<td>879.8 mA</td>
</tr>
</tbody>
</table>

Figure 5.2 shows a screen shot from oscilloscope with current consumption waveform when a WARP board with single radio daughterboard is transmitting packets with 1000 byte payload with different modulation schemes. Coherently to our measurement in Table 5.2, we can observe that the current consumption while transmitting is much higher than current consumption while the radio is in receiving state. The time taken to transmit equal length packets is almost double when BPSK modulation scheme is deployed as compared to QPSK.
Parameter Adaptation

Camp and Knightly have observed that for each modulation scheme, there is a range of SNR at which highest throughput is achieved [170]. Here, we measure in terms of packet delivery ratio with one single transmitter and one receiver with different interference level and payload modulation rate to derive a scheme for automatic rate adaptation with respect to energy consumption and throughput. The interference is generated using the vector signal generator as an unmodulated signal at the frequency band where the transmission reception path between WARP boards are established (5580 MHz) with amplitude varying from -20 dBm to 5 dBm. We have measured that the attenuation between the signal generated and the signal detected by the WARP radio frontend is 50 dBm, i.e. signal generated with 0 dBm amplitude creates an interference level at -50 dBm. The antenna distances between transmitter and receiver, transmitter and signal generator, receiver and signal generator are fixed to be 10 cm, 5 cm and 5 cm, respectively.

From Figure 5.3, we can observe that BPSK and QPSK delivers equal amount of packets when interference level is below -60 dBm. Between -60 dBm and -58 dBm, BPSK offers approximately 20 % to 30 % delivery performance gain while when the interference level goes above -58 dBm, the advantage of BPSK over QPSK diminishes to less than 5%. QAM16, while being very energy efficient and has the capability in offering high throughput in ideal environment, performs poorly in normal office environment in non-congested 5 GHz ISM channel on WARP boards.
Optimization Method  In this experiment, we use the lexicographic method [169] for optimization. With the lexicographic method, the objective functions are arranged in order of importance. Then, the following optimization problems are solved one at a time:

\[
\begin{align*}
\text{Minimize } & F_i(x) \\
\text{subject to } & F_j(x) \leq F_j(x'_j), j = 1, 2, ..., i - 1, i > 1, i = 1, 2, ..., k.
\end{align*}
\]

where \( k \) is the number of optimization objectives, \( i \) represents preference level of the objective function, and \( F_j(x'_j) \) represents the optimum of the \( j \)th objective function, found in the \( j \)th iteration. We incorporate our experimental readings as a priori knowledge for the runtime optimizer. We set the packet delivery ratio requirement to be 60\% and are interested in best energy consumption performance. Our optimization problem becomes the following:

\[
\begin{align*}
\text{Minimize } & E(x) \\
\text{subject to } & D(x) \geq 0.6.
\end{align*}
\]

Figure 5.4 shows the behaviour of a simple protocol which transmit packets once every 10 ms. Our optimizer checks the RSSI reading every time before packet transmission and chooses the modulation scheme based on the
current signal strength which gives satisfactory packet delivery performance with minimum current consumption. These facts are important to be used as prior knowledge for the optimizer for possible optimizations. Figure 5.4 shows that when the interference level is detected to be below -70 dBm, packet transmission using QAM16 is used as it is the most energy efficient option based on our measurements in Table 5.1 and Table 5.2. As the interference level goes up, more robust modulation schemes are needed to deliver the desired successful packet delivery ratio.

**Runtime MAC Protocol Composition**

In order to show the performance enhancement enabled by flexible runtime protocol composition, we measure performance of different MAC protocol behaviours under the same spectral environment. Figure 5.5 and Figure 5.6 show the performance of ALOHA, CSMA and Spectrum Agile MAC protocols achieved on WARP board with varying level of interferer occupancy ratio, measured between a single transmitter receiver pair. The implementation of the three protocol behaviours is as shown in Figure 4.3. The interferer occupancy ratio refers to the percentage of the duration per time period when a particular channel is being used by an interferer or other networks sharing the same spectrum. Our measurements show that with single channel single transmission flow, when the interference occupancy ratio is below 5%, ALOHA gives the best performance in terms of throughput while the packet delivery ratio is
almost on par with CSMA and Spectrum Agile MAC protocols. The packet delivery ratio for ALOHA drops significantly when the interference occupancy ratio is above 5%. When the interferer occupancy ratio is between 5% and 10%, ALOHA performs still the best in throughput but worst in packet delivery ratio. CSMA MAC gives a balanced performance in this region. When the interferer occupancy ratio goes above 10%, both ALOHA and CSMA start to perform poorly. Spectrum Agile MAC does not offer a high level of throughput when the interferer occupancy ratio is low due to the overhead induced by spectrum scanning and hopping. However, it has the ability to find and use a less interfered channel when the current channel is severely interfered. When the interferer occupancy ratio is above 10%, the performance of Spectrum Agile MAC protocol is much better than the other two protocols since it always finds a vacant channel for data transmission. In our implementation, the overhead for switching channel and establishing link between the transmitter/receiver pair causes a 5% drop in throughput assuming the channel switching occurs once every second, i.e. each channel that is found free stays free for one second. If the channel condition is very dynamic and channel switching is performed more frequently, the MAC performance of Spectrum Agile MAC will degrade. Either more efficient channel switching algorithm is to be used or staying in one channel using simple ALOHA or CSMA protocols might become more beneficial.

Figure 5.5: Packet delivery ratio of ALOHA, CSMA and Spectrum Agile MAC protocols running on WARP.
5.2. Runtime Optimization

In this experiment, we use a weighted sum method for optimization:

\[ U = \sum_{i=1}^{k} w_i F_i(x), \]  

(5.4)

where different weights \( w_i \) are assigned to different objective functions. In our case, the optimization goal is to maximize a combination of throughput and packet delivery ratio by giving 70% weight to throughput and 30% to packet delivery ratio:

\[ U = 0.7 \times S(x) + 0.3 \times D(x). \]  

(5.5)

The interferer occupancy ratio in a particular channel is varied from 0% to 80% as shown in Figure 5.7 while the rest of the 5 GHz spectrum is kept free. The protocol at compile time is composed with ALOHA-like behaviour since assuming the channel is free, it gives the best throughput performance. Unlike pure ALOHA, we use \texttt{ReadRssi()} in our protocol realization since it is necessary to monitor the channel condition as an on-line input to the optimizer. Performance feedback in terms of throughput is fed back to the optimizer. The throughput was observed to be high when the channel is free and experienced an immediate drop when interferer is detected. When the interferer is first detected at 5th second, the \texttt{CarrierSensing()} is added to ensure a reasonable level of data delivery performance while not losing too much throughput. When the interferer has left the channel, the \texttt{CarrierSensing()} com-
ponent is removed to increase throughput. For the first 15 seconds, we see a fluctuated performance as the \texttt{CarrierSensing()} component is added and deleted from the protocol since the interferer appears only for short periods of time. When the interferer occupies the channel for a long enough period of time, the \texttt{SelectChannel()} element is included since the confidence level of the current operating channel is low and a new channel should be searched for. When a free channel is found, good channel condition is experienced. \texttt{SelectChannel()} and \texttt{CarrierSensing()} are gradually removed from the protocol and the throughput performance is regained. In our implementation, the protocol adaptation is triggered based on a moving average of the observed RSSI and throughput value to avoid unnecessary reconfiguration due to extremely sporadic interference signals. We observe a performance gain up to 400\% at the end when the interferer occupancy ratio is up to 80\%.

5.3 A Compiler Assisted Scheme for Protocol Optimization

With a growing number and complexity of wireless communication applications, new challenges have arisen in terms of providing the required stable and high QoS due to the ever crowding spectrum, changing application requirements and the need to coexist with other wireless applications. In Section
5.3. A Compiler Assisted Scheme for Protocol Optimization

In Section 5.2, we have presented a runtime optimizer which together with TRUMP is able to carry out MAC adaptation based on performance optimization on-the-fly. The performance metrics for optimization such as throughput and packet delivery rate are specified before the deployment of the MAC scheme. However, in real-life applications, the metrics for the optimization can change over time. For instance, when a network is supporting a video streaming application, maximizing the data rate would be the priority while when the user sends an email over the network, data reliability is highly important. Therefore, in order to achieve the desired performance characteristics at all time, the application should be allowed to specify the preferences and modify the preferences on-the-fly. Different MAC metrics for optimization result in different MAC reconfigurations at runtime.

TRUMP has its own MAC meta-language for fast MAC protocol prototyping and a meta-compiler which converts the MAC protocol description into executables for the target platforms. In order to enable runtime protocol realization tailored to a wide range of user-specified application preferences, we have extended the capability of the MAC meta-compiler. We have adopted an approach similar to the car navigation system. The car navigation systems plan optimal route based on the position of a car in real time. The driver can specify a wide range of preferences and constraints such as minimum cost, minimum time, minimum distance, etc. The optimum solution is concluded based on the prior knowledge of the length of all the route, the toll costs and the traffic condition. It allows runtime update, route recalculation as the car position and driver preference changes. We have allowed the application to specify multiple criteria for MAC realization such as maximum energy efficiency, minimum latency and maximum data reliability through the meta-compiler. The meta-compiler requires pre-fed knowledge such as the execution time of each individual functional component, energy consumption at different protocol and hardware states, etc. Using this information, the meta-compiler selects the optimum combination of components and parameters.

5.3.1 System Design and Implementation

Figure 5.8 shows the architecture of our compiler assisted approach for protocol optimization using TRUMP. The optimization criteria which represents the application preferences is extracted by the MAC meta-compiler and passed to an optimizer. The MAC component performance file is queried by both the compiler and the optimizer at both MAC design stage and at runtime. The optimizer decides on the configuration and re-configuration of MAC protocols and passes its decision to the Wiring Engine of TRUMP to realize MAC protocols based on different optimization criteria. Only two criteria are shown in the figure for simplicity. At the design stage, the meta-compiler can provide the MAC protocol designer with the predicted protocol performance in terms of execution speed and power consumption based on the prior knowledge of
the MAC components without having to deploy the MAC code on the target platform. The compiler also selects MAC parameters based on designer specified preferences at the design stage by making use of the a priori knowledge of all the MAC components.

![System architecture of a compiler assisted approach for protocol optimization.](image)

Figure 5.8: System architecture of a compiler assisted approach for protocol optimization.

5.3.2 Meta-compiler Architecture

The MAC meta-compiler was initially implemented as part of TRUMP to convert protocol description written in our MAC meta-language to executables on the target platform. It prevents potential mistakes in protocol implementation and reduces the protocol development time as we have briefly described in Section 4.1.4. In this section, we elaborate on the compiler design which is enhanced for application optimization criteria selections. Our compiler is designed using Lex&Yacc [154] and runs on the target platform. The protocol develop-
The MAC description is passed to the meta-compiler. The application optimization goal which is described in detail in Section 5.3.3 is specified at the beginning of a MAC protocol description. The meta-compiler then converts the MAC description into the protocol list which is in the linked list form. The protocol list is directly executable on the target platform.

The compiler consists of three parts: a scanner to scan the program file to recognize keywords and tokens, a parser to determine the grammatical structure and checks for syntax error and a code generator which generate executable code accordingly for the target platform. There are two basic functionalities of a compiler: converting MAC description to executable code and handling the variables. Lex is used to implement a scanner which reads the input text and converts strings to tokens; then a parser built using Yacc maps tokens to the instruction set in Wiring Engine and MAC component library. Grammar rules are defined in the parser. Syntax error will be reported at the time of compilation. With the input tokens from scanner, the parser creates nodes for the linked list accordingly as shown in Figure 5.10.

### 5.3.3 Optimization Options

Standard GCC compiler has optimization options with respect to code size and execution time at different levels. Since our compiler is designed for MAC protocol realizations, our interest mainly lies in network performances and hardware constraints. The optimization options we have enabled through the compiler is shown in Table 5.3, which are the same as we have defined in Section 5.1. Compiler requires knowledge about execution time of each individual functions, energy consumption at different protocol and hardware states. Robustness of data delivery and one-hop throughput using different MAC me-
Table 5.3: Compiler optimization options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Optimization Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-e)</td>
<td>Energy consumption</td>
</tr>
<tr>
<td>(-s)</td>
<td>Throughput</td>
</tr>
<tr>
<td>(-d)</td>
<td>Robustness of data delivery</td>
</tr>
</tbody>
</table>

Mechanisms and parameters should also be measured. These optimization options captures the application requirements which might change during the course of protocol execution.

If there are multiple implementations for the same functionality available and the user does not specify a choice, the compiler selects the implementation which is optimized for user preferences. For example, the user may specify the modulation scheme used on the packet payload for packet transmission. As a simple example shown in Figure 5.10, if the designer wishes to achieve the highest throughput one can set the optimization goal to be \(-s\) which maximizes the throughput. The designer is not required to have any knowledge of the relationship between modulation scheme and throughput in order to achieve the design goal. When using the `SendPacket()` function, the designer can specify `ANY` for the modulation scheme. The scanner of the compiler scans the line
SendPacket(0x01, 100, ANY). It identifies SendPacket to be a function and 0x01, 100 and ANY to be parameters. Among the parameters, 0x01 and 100 are constants which will be parsed directly while ANY is identified to be a special parameter which requires the parser to query the MAC components performance profile which is included in the Resource Table with the optimization options specified. The parser looks up the table which contains all the options for this parameter (BPSK, QPSK, QAM16) and the performance metrics associated with these parameter options. The query returns QAM16 for our platform or any other modulation scheme which results in the highest data rate. QAM16 appears in the node for the protocol list for execution. If the user specifies maximum data reliability as the top priority, the compiler will return BPSK for the modulation parameter instead.

At runtime, an optimizer monitors the performance of the currently executing MAC. With feedbacks from the meta-compiler, it adapts the protocol according to the preference set by changing protocol functionalities and parameters. At the same time, it also monitors the spectral conditions by sampling received signal strength indicator periodically. For example, when a particular wireless channel is used by other networks, chances for the medium to be free are low. The optimizer will be aware of the increasing latency of delivering a packet due to the lack of opportunity to send packet in a free channel. If the application gives minimizing latency high priority, the protocol can deploy functionalities for multiple-channel support. It essentially provides a higher possibility within a time frame to locate a non-occupied medium for the packet to be delivered \cite{96}. However, using multiple channels leads to additional control overhead and message exchanges which results in more energy consumption. Therefore, if the application prefers to be highly energy saving, the protocol can choose not to adopt the multichannel scheme. In conclusion, based on the selected criteria and the spectral condition, the MAC protocol adapts itself to the best suited behaviour by modifying composition of MAC components at runtime.

### 5.3.4 Evaluation Results

We have implemented our solution on WARP v1 board based on the v16 OFDM reference design. Similar to the runtime optimizer described in Section 5.2, we need to measure the protocol performance with different MAC strategies and parameters under different spectral conditions and feed these pre-gathered knowledge to the runtime optimizer and the resource table for the compiler. The experimental setup is the same as described in Figure 5.1. One transmitter receiver pair of WARP boards are used for our measurements while the vector signal generator is used to generate interference. Figure 5.11 shows the part of the MAC performance in terms of energy consumption per packet delivered, packet delivery rate and throughput of the transmitter receiver pair with different MAC parameters and strategies under different spectral conditions.
Figure 5.11: Performance measurements of various MAC strategies and modulation schemes.
Figure 5.12: MAC performance statistics with different optimization goals.

We have demonstrated our tool at SIGCOMM 2012 [166]. The demonstration included two WARP boards connected to a PC. An interactive GUI is used to display live MAC performance statistics in terms of throughput, packet delivery rate and power consumption. A TelosB node is used to generate interference at the operating channel of the WARP boards at 2.4 GHz ISM band. The audience is able to modify the protocol optimization goal on-the-fly and correspondingly observe the triggered reconfiguration and the resulting MAC performances. Figure 5.12 shows the changing MAC statistics when the optimization goal is modified at runtime. The packet payload size is 100 bytes. We
see that when the optimization criteria is set to be data delivery, almost 100% successful data delivery ratio is achieved under all interfering environment. However, at the same time the energy spent per successfully delivered packet is significantly higher than in other cases when the interferer occupancy ratio is high and the throughput is slightly lower. The MAC performances with energy optimization and throughput optimization are the same. It is due to the fact of the little difference in terms of power consumption of WARP board in transmission mode and reception as show in Table 5.2. The total energy consumption per period time does not vary much base on the radio state. Therefore, the energy consumption per packet delivered is the lowest when most number of packets are delivered, thus coincide with the throughput. For a platform which has a large difference in power consumption among radio states, the MAC protocol reconfiguration trend can be different from Figure 5.12.

5.4 A COOPERATIVE SCHEME FOR RUNTIME MAC PROTOCOL OPTIMIZATIONS

As described in previous sections, TRUMP has enabled flexible adaptation of MAC behaviour on-the-fly resulting from spectral and network dynamics, mobility and varying QoS application demands. We have shown performance enhancements resulting from runtime MAC optimization on WARP boards with a transmitter receiver pair. The scenarios we have considered so far are one hop network where the interferer can be seen by all the nodes in the network and the optimization of MAC performance is carried out at each node individually. In this section, we expand our view to multi-hop networks. In a multi-hop network, runtime reconfiguration at individual nodes may lead to possibly incoherent MAC schemes in a neighbourhood. Especially when the spectral environment varies drastically spatially within a network, greedy local optimization might lead to loss of communication link among nodes. Therefore, in order to enable efficient network wide optimization, we introduce a cooperative mechanism using TRUMP where coexistence of different MAC protocols is allowed in the same network without disruption in communication links. The architecture of the cooperative scheme is designed based an intelligent agent concept [171] which is often used in an uncertain, dynamic and continuous environment as in our network environment. The cooperative scheme allows the neighbouring nodes to adapt their MAC behaviour in order to maximize the performance characteristics. This is achieved through a controlling mechanism, which monitors the MAC behaviour of its neighbours and disseminates this information to the network. The overhead caused by cooperation message exchanges is minimized by compressing MAC behavioural change into the packet header.
5.4. A Cooperative Scheme for Runtime MAC Protocol Optimizations

5.4.1 System Architecture

Based on the intelligent agent concept, we have designed and implemented a cooperative module. The cooperative module is plugged into TRUMP which can be easily enabled and disabled through a knob. The system architecture is shown in Figure 5.13. The MAC Adaptation Module refers to the part of the system which controls the runtime adaptation of MAC protocols, including the runtime optimizer and the compiler. The adaptation can be triggered by the spectral environment, application requirements and user preferences. The Communication Module is the interface for sending and receiving messages through the radio frontend. The Cooperative Module has the following functionalities:

- It keeps track of the MAC state of the node and its neighbours.
- It analyzes the re-configuration of MAC protocol triggered by the MAC Adaptation Module and extracts the differences between the current MAC protocol and the to-be executed MAC protocol.
- It decides if and what configuration should carried out based on the MAC states of the neighbouring nodes.
- It informs the neighbouring nodes of the changes of the local MAC state.

The Cooperative Module interacts with the MAC Adaptation Module to extract the changes required by the adaptation module and uses the Communication Module to exchange cooperative information among nodes. The Co-
operative Module contains several parts to carry out the functionalities listed above. The *MAC State Table* stores the description of the currently executing MAC protocol of the node and its neighbours. Instead of storing the entire MAC description file in the MAC State Table, we extract the MAC states from the protocol description. The extraction enables easy comparison of MAC states among nodes as well as saves memory usage. The MAC state is represented in terms of functionalities used by the MAC protocol. For example, if `ExpectingFrame(ACK)` has appeared in the MAC description, Acknowledgement functionality is extracted and stored in the MAC State Table. Acknowledgement is seen as a feature of the MAC protocol. Other features include channel selection, carrier sensing, etc. The mapping of function to feature is done using the *MAC Resource Inquiry Table*. The MAC Resource Inquiry Table contains all the functions of the MAC components with possible parameters. The features associated with each of the function are included in the Inquiry Table. The MAC state information of the neighbouring nodes are exchanged through either dedicated cooperation packets or piggy-backed to normal data packets. The *Cooperative Data Unit* is in charge of the extraction of cooperative information from received packets. It also prepares packet to disseminate the local MAC state to the neighbouring nodes. In addition, the Cooperative Data Unit compresses the MAC adaptation message to be processed by the *Cooperative Logic Controller*. The Cooperative Logic Controller acts as the brain of the Cooperative Module.

Figure 5.14 shows the operation flow of the Cooperative Logic Controller. Node initialization is carried out when a node is powered up and joins a network. It fills the local MAC information into the MAC State Table and broadcasts a cooperative packet containing its MAC information to its neighbours requiring their MAC state information. The neighbouring nodes sends acknowledgements with their local MAC states. When a packet is received, it is analyzed to see if cooperative information is contained. If yes, the MAC state information will be extracted and the MAC State Table will be updated. MAC adaptation will be carried out if required through the received cooperative information. MAC protocol reconfiguration can also be triggered by the MAC Adaptation Module. The Cooperative Logic Controller first compares the MAC adaptation requirements to the current MAC state to decide if there is a change in MAC state. For example, if the adaptation requires a change in the carrier sensing duration, there will be no change in the MAC state. However, if a new line `setFrequency()` is added, the MAC state will be changed since channel selection is one of the features we have identified for MAC state description. The local MAC state will then be compared to the neighbouring nodes to see if the changes in MAC state should be announced. If needed, the new MAC state will be included in a data packet or a cooperative packet to be sent to the rest of the network. The cooperation ensures that the communication link among nodes are stable even if the nodes are experiencing extremely different environmental changes.
5.4. Evaluation Results

We have implemented the cooperative scheme and integrated it with TRUMP and Decomposable MAC Framework. The implementation is done base on OFDM reference design v16.1 on WARP board v1. The cooperative scheme introduces execution overhead to the network due to the node initialization process and cooperation message exchanges. In this section, we present the execution overhead of these processes in several network topologies measured on WARP boards. All the data packets used in our experiments are modulated with BPSK with a 100 byte payload size. All the cooperative packet is of zero payload since all the MAC state information is compressed to the MAC header.

Table 5.4 shows the execution delay introduced by various process of the cooperative scheme in network with different topologies. The initialization process refers to the procedure of broadcasting a cooperative packet to the network, receiving all the acknowledgements from the network and filling the MAC State Table with the information gathered. Cooperation with cooperative packet refers to the during between when MAC adaptation requirement
Table 5.4: Execution delays introduced by the cooperative scheme to different MAC processes and network topologies.

<table>
<thead>
<tr>
<th>Network</th>
<th>Process</th>
<th>Latency [us]</th>
</tr>
</thead>
<tbody>
<tr>
<td>One node</td>
<td>MAC State Table initialization</td>
<td>218.7</td>
</tr>
<tr>
<td>One node</td>
<td>Broadcast initialization packet</td>
<td>395.1</td>
</tr>
<tr>
<td>One node</td>
<td>Write MAC state to packet</td>
<td>19.0</td>
</tr>
<tr>
<td>One node</td>
<td>Read MAC state from packet</td>
<td>18.2</td>
</tr>
<tr>
<td>One node</td>
<td>MAC State Table update</td>
<td>230.6</td>
</tr>
<tr>
<td>Two-node one-hop</td>
<td>Initialization process</td>
<td>627.9</td>
</tr>
<tr>
<td>Two-node one-hop</td>
<td>Cooperation with cooperative packet</td>
<td>883.3</td>
</tr>
<tr>
<td>Two-node one-hop</td>
<td>Cooperation with data packet</td>
<td>1015.3</td>
</tr>
<tr>
<td>Three-node one-hop</td>
<td>Initialization process</td>
<td>856.8</td>
</tr>
<tr>
<td>Three-node one-hop</td>
<td>Cooperation with cooperative packet</td>
<td>1293.7</td>
</tr>
<tr>
<td>Three-node one-hop</td>
<td>Cooperation with data packet</td>
<td>1428.1</td>
</tr>
<tr>
<td>Three-node two-hop</td>
<td>Initialization</td>
<td>1670.3</td>
</tr>
<tr>
<td>Three-node two-hop</td>
<td>Cooperation with cooperative packet</td>
<td>2230.5</td>
</tr>
<tr>
<td>Three-node two-hop</td>
<td>Cooperation with data packet</td>
<td>2532.1</td>
</tr>
</tbody>
</table>

is received at the Cooperative Module till all the acknowledgements are gathered and the MAC adaptation request is forwarded to the Wiring Engine to be carried out. A dedicated cooperative packet is generated for information propagation. The cooperation process takes longer time than the initialization process due to the need to analyze the reconfiguration command, comparing local MAC state to MAC states to the neighbouring nodes, etc. Cooperation with data packet refers to the same duration as cooperation with cooperative packet except that instead of a dedicated cooperative packet a normal data packet is used for cooperative information propagation. The reserve bits in the MAC header of the data packet are used to store the MAC state information. The execution delay using data packet is longer than using cooperative packet due to longer transmission time required for a larger packet. The delay can be further extended since data packet availability is uncertain. Although using data packet adds an unknown latency to the cooperative information propagation, it reduces the packet exchange overhead of the network. Overall, we see that the cooperation delay is a couple of milliseconds which does not introduce significant negative impact on MAC performance as our results indicated in Section 5.4.3.

5.4.3 Demonstration

We have also demonstrated the our cooperative scheme in MobiSys 2012 [168]. We used four WARP boards in our demonstration to form a two-hop network. The boards are connected to a PC to display the MAC performance at each
node. A GUI is provided for the audience to interactively change the protocol configuration at individual nodes, spectral conditions and/or the network topology and correspondingly observe the resulting performance characteristics. We have shown that our cooperative scheme enables MAC reconfiguration in a multihop network neighbourhood. The scheme allows nodes to reconfigure their MAC behaviour with a low controlling overhead while being able to communicate seamlessly even when a node joins or leaves the network. Different topologies and node mobility are realized by modifying the transmission power and receiver sensitivity of the nodes to control the transmission ranges. Multiple interferers are introduced to the network at different physical location to model a dynamic spectral environment. TelosB nodes are used for interference generation at 2.4 GHz.

In this section, we select one scenario from our demonstration setup as shown in Figure 5.15 to measure the MAC level goodput of the network. In our network layout, the transmitter (TX) node shares spectrum with all three receivers while receiver node RX1 is out of transmission ranges from other receiver nodes RX2 and RX3. A local interference is generated to affect the transmitter and RX1 only to create a difference in the spectral environment for nodes within the same network. The runtime adaptation logic of all nodes is the same as described in Section 5.2.2.

Figure 5.15 shows the MAC level throughput of all the nodes in this two-hop network with varying level of presence of the interferer. We can see that for transmitter node TX and receiver node RX1 which are affected by the interference, they behave according to the optimization rules governed by the runtime optimizer as described in Section 5.2.2. At approximately 16s, when the interference occupancy ratio exceeds 20%, both nodes switch to spectrum agile MAC and find another channel with spectrum availability for transmission. Without the cooperative scheme, receiver nodes RX2 and RX3 are not aware of the change in spectrum condition, and thus retains their MAC behaviour and fail to receive more packets.

Figure 5.15: Experimental setup of a two-hop network for cooperative scheme measurements.
Figure 5.16: The throughput performance of all nodes in the two-hop network shown in Figure 5.15 with and without using the cooperative scheme.
5.5. Summary and Discussions

With cooperative scheme, the transmitter will communicate with all receivers within its vicinity when a crucial behavioural change in MAC is required. Since as shown in Section 5.4.2, the delay caused by cooperative packet exchange is only approx. 2 ms for a three-node network, the time required for cooperative communication in our four-node network is also not significant. We can observe an almost negligible drop in terms of throughput at 16 s for TX and RX1 when the cooperation mechanism is activated. Both RX2 and RX3 are informed about the change that TX is making and they adapt accordingly. Therefore, a seamless communication is achieved within a highly dynamic spectral environment while the cooperative scheme handles control information exchanges.

5.5 Summary and Discussions

New challenges are arising in providing sustainable QoS in wireless networks with a growing number and complexity of wireless communication applications. The challenges are mainly due to the ever crowding of wireless spectrum, changing application requirements and the need to coexist with other wireless applications. Since the spectral environment and the application performance expectation can be highly dynamic, a static MAC protocol is not able to suffice the application requirement at all time. We have introduced TRUMP in Chapter 4 which allows runtime protocol realization and re-configuration. Having the ability to realize a vast range of MAC protocols at runtime, we focus on optimization of MAC behaviour to provide desired MAC performance on-the-fly.

In this chapter, we have introduced a runtime MAC optimizer which uses TRUMP for MAC adaptation. In order to optimize the performance at the MAC layer, we monitor the network environment, application parameters and runtime MAC performance feedback at the same time. Our experimental results show that with a runtime MAC optimizer, we have enabled flexible MAC adaptation both in terms of parameter tuning and functional re-composition. Our MAC protocol is able to provide a performance gain in terms of throughput up to 400% in a severely interfered environment in our test case. In order to enable runtime protocol realization tailored to a wide range of user-specified application preferences, we benefit from a MAC meta-compiler which is part of TRUMP. The application can impose multiple criteria/parameters for a MAC realization such as maximizing energy efficiency, minimizing latency and maximizing data reliability through the MAC meta-compiler at runtime. The meta-compiler interacts with the runtime optimizer to select the most appropriate MAC parameter and functions according to the optimization criteria. We have demonstrated different MAC adaptation behaviours under different application preferences. To ensure the node specific local optimization of MAC protocol does not bring detrimental effect to the overall network performance,
such as loss of communication between nodes due to extremely different spectral conditions experienced individually, we have implemented a cooperative scheme which is integrated with TRUMP. The cooperative scheme keeps track of the MAC states of all the neighbouring nodes and triggers necessary MAC state exchange when a MAC adaptation is to be performed at runtime. Our results show that a seamless communication is achieved within a highly dynamic spectral environment in a multi-hop network where the cooperative scheme handles control information exchanges.
MAC procedures for emerging network technologies demand fine grained timing control, high computational power and adaptability. MAC layers for high data rate standards such as LTE and IEEE 802.11ac require fast real-time response. Instead of the traditional single mode terminal which is optimum for performing a specific task, multi-mode system which supports multiple wireless protocols is required for 4G, the next generation of wireless communication systems [16, 172] and software defined radios. The need for computing power is drastically increasing for wireless terminals due to the increasing complexity of spectral condition and application requirements. Sophisticated sensing mechanisms and channel selection algorithms are often needed to fully utilize the available spectrum and to provide desirable level of services. In the context of Cognitive Radios (CRs) paradigm, often machine learning techniques and statistical data analysis algorithms are employed for runtime resource management, channel allocation at MAC layer, and performance optimization based on PHY- and MAC parameters. Machine learning algorithms are typically computationally intensive and a short convergence time is desired for rapid runtime adaptation.

In order to provide the flexibility for the new generation of wireless protocols, flexible computation structures are needed. SDR platforms emerge to be an attractive option for fulfilling the demands of great degree of flexibility. There has been a lot of research effort on the physical layer in this respect [173-175] but very few on MAC layer research. SDR platforms are typically equipped with computationally powerful units such as General Purpose Processors (GPPs) to handle the baseband modem signal processing requirements while offering a high level of flexibility. However, performance results indicate that this approach has shortcomings in meeting strict timeliness and scheduling requirements of MAC processing. Lee and Mudge [176] have shown that a GPP, while flexible enough, is difficult to provide the real-time requirement imposed by high data rate applications and MAC protocol standards while coprocessor-FPGA architectures, have shown to outperform the GPP based protocol processing units. In order to efficiently meet the computational load of evolving high data rate standards, multiprocessor architectures are proposed to achieve parallelization gains for baseband signal processing. The current multiprocessor architecture trend is shifting from multi-core to

MAC PARALLELIZATION ON MANY-CORE ARCHITECTURE
Ron Wilson has pointed out that there is a shift in the concept from making one processor powerful enough to handle all the data towards a more distributed approach of many processors sharing the work [177]. While heterogeneous architectures are commonly used and are known for their power efficiency, flexibility and reconfigurability offered by them remain limited as large implementation and debugging efforts are needed whenever extensions or modifications are made [17]. Homogeneous architecture, on the other hand, makes it easier for a software designer to map the applications, threads, parallel tasks, onto any processors in a flexible way as no special care needs to be taken, like in the case of for a heterogeneous architecture, different types of processors and hardware accelerators. The required implementation efforts and the resulting performance characteristics also heavily depend upon the tools and software support provided by an SDR platform. Highly dynamic MAC protocols, especially for CR environments, require specialized tools and frameworks to efficiently schedule tasks and manage radio resources at runtime in order to achieve desired reconfiguration and adaptation.

In this chapter, we experiment with parallelization of two major classes of MAC protocols on a) a multi-core x86 Linux based machine, and b) a many-core computing fabric P2012 [178]. Two main types of MAC procedures are considered for evaluation: classical CSMA based MAC protocols and reconfigurable MAC schemes using machine learning based algorithms for runtime optimization and resource management. TRUMP is used for MAC process parallelization and scheduling. Moreover, we discuss how SDR architectures with homogenous computing elements can lead to great degree of flexibility while satisfying the real-time requirements for MAC processing. This discussion has its peculiar importance as it discusses an alternative to the current belief in heterogeneous SDR architectures.

An outline of the rest of the chapter is as follows. In Section 6.1 we describe experimental results of MAC parallelization on Linux based x86 multi-core GPP platform. The promising initial results motivated us to further investigate into the hardware platform architecture for MAC process parallelization. In Section 6.2, we review different SDR platform architectures used by the community and how they perform in terms of flexibility, scalability, and programmability. We also introduce in detail our test platform P2012 in detail and the software tools support including TRUMP. We have implemented two main categories of MAC schemes on P2012 platform. We present the evaluation results in Section 6.3. Finally, we conclude in Section 6.4. This chapter is mainly based on our articles [151, 179] which were published during the dissertation work.
6.1 MAC PARALLELIZATION ON X86 BASED MULTI-CORE PLATFORMS

Multi-core architectures are nowadays common in general purpose computing and in high performance computing. For wireless communication networks, multi-core platforms are becoming a promising alternative for reducing single core design complexity and power consumption. Since needs for parallelism has been identified for MAC protocol execution, we explore the possible performance improvement we can benefit from multi-core platforms. In order to investigate the benefits and drawbacks of multi-threading and parallelization in terms of execution speeds, we have used a x86 Linux based PC as our testbed. We use TRUMP for MAC protocol parallelization and scheduling since TRUMP has the capability to identify the dependencies among components and dictate the execution sequence of the MAC protocol. A thread-pool has been implemented as part of TRUMP which offers threads for the scheduler. The thread-pool implementation for the Linux based PCs is realized using the pthread library. TRUMP also uses the schedutils for setting processor affinity for a certain task. It allows for the experimentation of emulating different level of computational power availability by forcefully assigning tasks onto different number of CPUs. The timings measured on WARP board in Table 5.1 are used to give a more educated guess on how parallelization can benefit MAC protocol executions when the hardware platform provides the capability.

6.1.1 Measurement Setup

We use a Linux based PC with Duo Core 1.83 GHz, 2M Cache, 2048M RAM. A sample function library is built to resemble the components in Decomposed MAC Framework. The execution time of the components is modelled according to the measurements from WARP board. Different numbers of while-loops are used to realize functions with different execution time. We have defined three factors that may impact the execution efficiency of these component-based applications: the execution time of the functions, the number of threads and number of CPU cores used. For experiments which require more than two cores, a 16-core Linux based PC is used with 2.8 GHz processor frequency per core and cache size of 1024kB. Profiler GNU gprof, Version 2.20.90 [180] is used for our timing measurements.

6.1.2 Evaluation Results

We use a four-function protocol in this experiment to show the benefit of multi-core multi-threading environment on the simplest case of MAC protocol. The protocol and the parallel dependency of the components are listed in Algorithm 2. Figure 6.1 shows the execution time for the program using a single CPU when different numbers of threads are used. We use a monolithic imple-
Algorithm 2: Components and their interdependencies for measurements in Figure 6.1.

Radio_to_Tx();
WriteToTxBuffer();
ReadFromRxBuffer();
TxPacket();
Radio_to_Tx() || WriteToTxBuffer() || ReadFromRxBuffer();
WriteToTxBuffer() || ReadFromRxBuffer() || TxPacket();

Figure 6.1: Execution time with and without TRUMP on a single-core Linux machine with different number of threads.

The monolithic implementation is simply executing four functions in sequence. Since the CPU power is shared with other OS processes on PC, both monolithic implementation and single thread execution time are much longer (approximately two times more) than execution time on WARP. When more resources are allocated with larger number of threads, the execution time has been reduced by 62%. Given the protocol and parallel dependency defined above, it is calculated that the theoretical execution time with single thread is about 1.48 ms and with multi-threads execution it can be reduced to 1.44 ms. We can see from the figure that the single thread execution time with our system is a little longer than the monolithic one, which shows a small system overhead about 0.1 ms due to the component-based MAC protocol architecture and the execution tools by TRUMP. Note that in the sample protocol only three func-
tions are allowed to run in parallel, the performance improvement is not significant for more than two threads. This is due to both the short execution times of `Radio_to_Tx()`, `WriteToTxBuffer()` and `ReadFromRxBuffer()` as compared to `TxPacket()`, and the management overhead of having more threads.

Computationally intensive algorithms are used widely in the area of CRN for spectrum sensing and prediction. In order to efficiently utilize the available spectrum while minimizing interference to PUs, cognitive radios are expected to keep track of history of spectrum availability and make predictions based on the observation. Prediction algorithms based on statistical analysis can be used based on the knowledge of the spectrum usage information to predict the future profile of the spectrum \(^{149}\). In the following experiment, we use a 16-core Linux based platform to fully evaluate the benefit and overhead of multi-core support. We have assumed a situation where the spectral condition history of 16 channels are stored and to be processed. The processing of the data of each channel can be carried out independently, thus offering the opportunity for parallelization. We scheduled the 16 independent tasks (to compute mean and standard deviation over 1000000 samples) to different number of threads and processor cores. Figure 6.2 shows the execution time of this sample application using \(n\) threads running on \(n\) cores. This measure-

![Figure 6.2: Execution time of a statistical process on a large set of 16 groups of data as 16 independent tasks on a 16-core Linux machine on different number of cores.](image)
ment is carried out for evaluating the benefit of multi-core architecture since only one thread is allocated to one processor core. The result shows that from using one core to four cores, the execution time has been reduced drastically from about 260 ms to 60 ms, and a further reduction of 35 ms as the number of cores used increased to 16. The execution speed has been reduced by 90% when using 16 cores as compared to a single core, which shows a large benefit of the parallelization supported by TRUMP on multi-core platforms.

Figure 6.3 shows the execution time of the same sample application using different number of threads on different number of processing cores. Multiple number of threads are used on a single core in this experiment. The optimum performance is achieved when 12 threads are used on 12 cores for execution of 16 tasks instead of 16 threads on 16 cores which would be the expected optimum. It is due to the overhead for thread scheduling. Therefore, it is not always the most efficient to use all the cores and threads available to execute tasks since the performance does not scale linearly with resource availability. It is an interesting optimization problem of matching the most suitable resource to the task demands.

![Figure 6.3: Execution time of a statistical process on a large set of 16 groups of data as 16 independent tasks using different number of threads on different number of processing cores.](image)

From these results, we have observed great potential in parallelizing MAC processes, especially for computationally intensive algorithms which are needed in new age of MAC protocols for spectrum sensing, channel allocation, runtime parameter optimization, etc. Our measurements results in this section
is highly dependent on the operating system used, since the OS introduces a significant amount of overhead in scheduling and background task processing. Furthermore, powerful CPU based multi-core architecture is not suitable for radio terminals mainly due to the high level of power consumption and size of the platform. Therefore, in the next sections, we discuss on the various platform architectures designed for embedded systems, especially radio terminals for software defined radio. We carry out experiments on a platform designed for power efficiency and flexibility which the results will reflect more closely to what one may expect from a mobile terminal which supports parallel execution of MAC processes.

6.2 PLATFORM ARCHITECTURE FOR EFFICIENT MAC REALIZATIONS

The platform architectures that we are discussing in this section are mainly developed in the SDR domain since the need for flexibility in radio technologies has resulted in a vast range of hardware architectures for mobile terminals with SDR functionalities. Various technologies such as Application-Specific Integrated Circuits (ASICs), FPGAs, Digital Signal Processors (DSPs), and GPPs, have been used in SDR platforms, commercially available products, military applications, and research prototypes. These technologies are used in both standalone and hybrid fashion [181]. In this section, we discuss the characteristics of different technologies and their implications on real-time MAC protocol performance and their suitability for emerging MAC protocols which demands high level of flexibility and computational power.

6.2.1 Task Specific Processing Elements

In standardized network interface cards such as IEEE 802.11 b/g where flexibility and reconfigurability requirements are limited, an ASIC is often used to handle the physical and MAC layer processes. ASIC solutions are highly optimized for realizing a particular computationally demanding protocol algorithms. They offer high level of computational efficiency and low power consumption. Although ASIC implementations are static and rigid, they can be suitable for implementing common functionalities across different configurations to accelerate the protocol execution speed and lower the power consumption [182]. The idea of implementing common and computational intensive functionalities in hardware for speed gains instead of pure software has also been proposed [183]. While many vendors provide standard compliant NICs, Bianchi et al. have shown that having programmability and reconfigurability at MAC layer helps in increasing the achieved throughput [39]. Application-Specific Instruction-set Processors (ASIPs) are typically tailored to a specific application and exhibit a lower energy consumption than GPPs or DSPs while
offering more flexibility than ASICs [184]. However, new standards demanding high flexibility, reconfigurability and multi-mode operation, make both ASICs and ASIPs alone not a very viable option for SDR platform implementation.

6.2.2 Reconfigurable Processing Elements

As compared to ASICs and ASIPs, FPGA is a reconfigurable solution at the expense of lower processing speed, higher power consumption and circuit area. There are several SDR development platform implemented based on FPGA. As an example, WARP boards [66] developed by Rice university are built using Xilinx Virtex FPGA and aim at offering flexibly PHY/MAC layer development. Computational intensive processes, signal processing are implemented in the FPGA while the application layer and some control functionalities are implemented in the PowerPC core in the FPGA. WiNC2R [185] is another example of a SDR platform built on a FPGA with soft-core processors and accelerators. Runtime reconfiguration of FPGA can be realized by partial reconfiguration, which requires significant efforts in FPGA development and is highly dependent on the tools and devices available. It can also be realized by software programmable reconfiguration, which has the limitation that all the program component has to be implemented before hand and the configuration options are limited to the controlling parameters which have been exposed to the soft-core. FPGA based architecture is more suitable for experimentation and prototyping than standardized commercially available SDR platforms due to the relatively slow processing speed. Furthermore, since the size of FPGAs is limited, it does not offer good scalability and is expensive to implement multiple computational intensive algorithms for MAC layer optimizations.

6.2.3 General Purpose Processors

Microprocessor systems provide full real-time programmability [186]. CalRadio [187] is a flexible wireless platform developed at UC San Diego targeting at fully programmable MAC protocols. It uses Intersil HFA3836 baseband chip for IEEE 802.11b PHY layer implementation which offers parameters such as the data rate and transmit power to be controlled by the MAC layer through register configurations. CalRadio provides a DSP for MAC layer implementation entirely in software which allows a high degree of flexibility in MAC layer design, though the packet transfer delay from host to DSP to PHY has limited the throughput to IEEE 802.11b PHY layer [188]. GPPs are typically unable to handle wideband signal processing in a timely manner to comply with the protocol standard. Therefore, multi-core architectures are introduced to achieve better performance by parallelizing processes. Parallel operations significantly reduce the execution speed and the power consumption per ins-
6.2. Platform Architecture for Efficient MAC Realizations

Sora [65] exploits parallelism in the MAC/PHY layer processing and is able to comply with IEEE 802.11 b/g standard. However, modification and extension to MAC/PHY implementations on Sora are highly complicated due to the sophisticated distribution of computational processes on multi-core processors in an effort to meet the real-time requirements. USRP1.0 [61] does the baseband signal processing are done on the host PC implemented using GNU Radio or National Instruments LabVIEW. The throughput achieved on USRP boards is typically little since the CPU processing power is the bottleneck. GPP based approach is good for fast PHY/MAC layer development. However, the processing latency and the power consumption are two major issues. Therefore, GPPs and/or DSPs often require hardware acceleration. Lau et al. have discussed the use of FPGA and ASIP based hardware accelerator in SDR waveforms and concluded that hardware accelerator enhances power efficiency which is essential in making SDR platforms into mobile terminals and handsets [189].

6.2.4 MPSoC Approach

Multiprocessor System-on-Chip (MPSoC) consists of multiple programmable processors. Heterogeneous multi-core architecture is popular for its power efficiency, high performance and low cost. IMEC’s baseband engine for adaptive radio (BEAR) platform consists of six cores (three ASIPs, one ARM processor, two architecture for dynamically reconfigurable embedded systems (ADRES) processors) and two accelerators [190]. Infion MuSIC-1 platform [191] is also a heterogeneous multi-core platform which consists of four programmable DSP cores and accelerators for Finite Impulse Response (FIR) filter, Viterbi decoder, etc. The heterogeneity allows different processes to be implemented on the most appropriate processor and thus achieve a speed-efficient solution. MAGALI platform [192] is a heterogeneous Network-on-Chip (NoC) based MPSoC platform dedicated for mobile terminals. This platform uses a centralized control processor for achieving power efficiency. However, this scheme limits the scalability of the architecture. The limitation applies to heterogeneous architecture in general. Moreover, as the complexity increases in heterogeneous architecture, often with irregular organization of memory hierarchy, efficient mapping of protocol algorithms on them is difficult [193].

Homogeneous many-core architecture provides a mid-way between multicore CPUs and Graphics Processing Units (GPUs) for a balance between programmability and parallelism. GENEPY (homoGENEous Processor ArraY) platform [17] is purely homogeneous, with Smart ModEm Processors interconnected with a NoC. Although homogeneous architecture is, in general, believed to be less efficient in speed and power consumption than heterogeneous architecture at the expense for offering easier programmability, higher flexibility and scalability, the authors have shown that for an LTE application, GENEPY has performance gains of 3% in speed and 18% in power consump-
tion as compared to MAGALI platform. In this chapter, we have used P2012 platform which consists of homogenous processing clusters for MAC protocol parallelization.

6.2.5 Platform 2012

In order to span the wide efficiency spectrum between fully programmable homogeous many-cores and application specific accelerators, a new family of computing systems called Many-Core Computing Fabrics (MCCFs) have been introduced. MCCF consists of many homogeneous processing cores interconnected by a NoC infrastructure [194]. Platform 2012 (P2012) is an area- and power-efficient MCCF developed by ST Microelectronics [178]. P2012 aims at filling the gap between general-purpose embedded CPUs and fully hardwired application accelerators in terms of area and power efficiency. It is flexible in supporting a wide range applications while not losing power efficiency. P2012 is based on four globally asynchronous locally synchronous (GALS) clusters. One cluster consists of a multi-core computing engine called ENCore and a Cluster Controller (CC). The EnCore cluster can host up to 16 Processing Elements (PEs). All EnCore PEs share a L1 tightly coupled data memory (TCDM) which supports a throughput one data access per PE per clock cycle. A low latency network is used to interconnect the PEs and on-chip shared memory banks within an EnCore cluster. A hardware synchronizer is used for ENCore to provide scheduling and synchronization acceleration. The hardware synchronizer also has a dynamic allocator which allows the system to dynamically allocate the best available PE to execute a task, which is suiting to the flexible and unpredictable nature of MAC processes in a dynamic environment. The CC takes care of booting and initializing the ENCore PEs and deploying applications onto the PEs. It consists of a Direct Memory Access (DMA) subsystem which transfers data blocks between the external memory and the internal memory during operation of PEs. The fast memory access facilitates MAC protocol realization especially in meeting the real-time requirements. Furthermore, the memory among different clusters are transparent on the platform, i.e. one cluster can directly access the memory on other clusters. No memory copying overhead is induced and therefore real-time MAC execution can be realized as MAC processes typically involve multiple data accesses with low-delay tolerances. These clusters are connected via a high-performance fully-asynchronous NoC. The clusters are implemented with independent power and clock domains, enabling aggressive fine-grained power, reliability and variability management.

Software Tools

There are mainly two layers in the software stack for P2012. The runtime layer interacts directly with the P2012 fabric and provides basic functionalities such
as task scheduling, dispatching, memory allocation, resource and power monitoring, host-fabric communication, etc., to upper layers. The programming model layer provides high level environment for developing specific programming models and applications. P2012 supports industrial standard programming models such as OpenCL and OpenMP programming models. We have developed our MAC layer schemes for P2012 using the Native Programming Model (NPM). The NPM allows developing specific applications running on a P2012 fabric and integrating them with the host system. The NPM is highly optimized for the P2012 architecture. It takes into account the specific features of the P2012 architecture like direct access to hardware synchronizer and DMA, or the partition between CC and ENCore Processors, thus providing the highest level of control on application-to-resource mapping at the expense of abstraction. Our MAC schemes fully utilize the NPM capability to achieve fast execution and provide fast response to the network. Applications for P2012 require execution engines to manage the interaction between CC and ENCore processors. Execution engines provide methods for initializing, starting, notifying and stopping ENCore PEs. The Reactive Task Manager (RTM) is an execution engine supported by NPM. RTM runs in a cluster and allows easy fork/join and duplication of jobs on PEs. The Multi-Thread Engine (MTE) is another execution engine available through NPM. The MTE uses threads to parallelize processes. Barriers are used for synchronization and the threads cannot be preempted. Multiple threads can run on either single or multiple PEs within one cluster. Using both MTE and RTM based on their different capabilities in mapping tasks to PEs, we have implemented a toolchain TRUMP for MAC protocol parallelization and scheduling.

**TRUMP on P2012**

TRUMP is a toolchain for runtime protocol realization. It consists of a MAC meta descriptor for MAC protocol design in C-like syntax, MAC meta compiler which interprets the MAC description for the target platforms, and Wiring Engine for managing the runtime execution of the MAC protocol. TRUMP aims at providing parallelization possibilities of independent MAC processes. It has a dependency table which captures the dependencies among different MAC processes, and a logic controller which governs the scheduling of the MAC processes based on the availability of thread/processor core and the state machine of the MAC protocol. We have used TRUMP on a x86 Linux based multi-core PC for simulation of some MAC protocol configurations as described in Section 6.1. A reduction in terms of execution speed of 90% is observed in our test case as compared to a single core single thread environment. We have implemented TRUMP on to P2012 platform using NPM and its execution engines for easy MAC protocol realization as shown in Figure 6.4. As part of TRUMP, the MAC meta-compiler is implemented on the host side which processes both the MAC description and the dependencies indicated by
Figure 6.4: System architecture of TRUMP implementation on P2012 platform.

The MAC designer for the functions. An execution list which contains the logic and functions used by the MAC description is formed. The list is analyzed and a two-dimensional array is written with the logic operator, functions and the dependency code associated. The array is passed to the Fabric Controller (FC) as an argument of a message. The FC uses the Wiring Engine to map the functions in the array to the MAC components in the library. Depending on the nature of the MAC layer applications, TRUMP uses different runtime execution engines. MTE and RTM can also be used together on different clusters. For example, standard MAC protocol processes uses MTE since individual thread terminates independently. RTM is more suitable for parallelizing of duplicated tasks and callback function is only triggered when all the tasks on the PEs within one cluster are done execution, which is suitable for some of the machine learning algorithms for MAC schemes.
6.3 MAC Parallelization on Many-core P2012 Platform

Since we have identified the advantage of homogeneous many-core computing fabric for future mobile terminals in terms of area- and power efficiency, flexibility, scalability and easy programmability, we explore the possible performance improvement on the P2012 platform using TRUMP. Since our platform does not have RF front-end, we are not able to execute real MAC protocols for data communication, etc. Therefore, similar to our approach in Section 6.1, we have implemented all the basic components in our MAC library according to the readings measured on WARP board in Table 5.1. In order to assess the benefit and drawback that many-core architecture brings for new generation of MAC protocols, we have implemented three types of MAC layer applications on a cycle accurate P2012 emulator for evaluation: a) classical MAC schemes which do not have significantly computational algorithms; b) genetic algorithm based runtime MAC performance optimization algorithm, and c) swarm intelligence based channel selection algorithm. Additionally, since multi-core architecture requires additional scheduling and management mechanisms, we also present the execution overhead for task scheduling and the initialization and termination overheads of the platform. The error rate of the cycle-accurate emulator is around 10%. Please note that our measurements are platform specific and the results will not apply exactly to other MCCFs. However, we believe the results are good indicators for the potential benefits we can obtain at the MAC layer using many-core platforms.

6.3.1 Classical MAC Executions

For our measurements in this section, we have used only one cluster which consists of 16 PEs on P2012. The Wiring Engine of TRUMP runs on the cluster controller and governs the scheduling of functions onto different Processing Entities (PE). As a simple test to measure the overhead caused by scheduling processes onto different cores, and at the same time compare the differences between using the P2012 many-core computing fabric and multi-threaded x86 Linux based PC, we use a list of four functions with parallel dependency exactly the same as used in Algorithm 2 in Section 6.1.2.

The results in Figure 6.5 show that there is a slight improvement in using more than one processing element to execute the above listed functions. Additionally, we have also measured the improvement on executing ALOHA and simple CSMA MAC protocols. The savings in terms of execution time are generally small, at around 3%. This is due to the limited opportunity for parallelization among MAC procedures and the dominating time consuming processes such as SendPacket() cannot be further parallelized. There is also scheduling overhead involved in MAC executions which we are going to investigate in details in the next experiment.
We have implemented a simple MAC protocol as shown in Table 6.1 using TRUMP. Similarly, timing measurements from WARP boards are used to emulate a more realistic MAC behaviour. Some of the functions are independent from others and can be executed in parallel, e.g., `BackOff()` and `SetFrequencyChannel()` while some functions have to be executed in a specific sequence, e.g., `SendPacket(ACK)` has to be executed after `WriteToTxBuffer(ACK)`, as the transmit buffer needs to be filled with the relevant data before a transmission should take place.

We have measured the complete execution time of the above described MAC protocol on P2012 using different number of PEs. Figure 6.5 shows the ratio of the execution time of parallelized MAC protocol using multiple PEs against sequential execution using one PE over varying number of iterations. Approximately only 3% of execution time has been saved by using parallelization for this MAC protocol realization. It is due to the high level of dependency of the function used in this protocol and the short execution time required by each functions. We have analyzed the complete execution time and identified two main parts contributing to the overhead of executing a MAC protocol on the P2012 platform. The initialization overhead includes the initial communication between the host and the fabric and the initialization of fabric. The scheduling overhead refers to the time taken for the CC to schedule tasks onto the PEs at runtime. There is always a delay between task executions due to the central controller. Even for independent MAC processes, there is a difference in the starting time since they need to be allocated onto the PEs in sequence. Figure 6.7 shows the initialization and scheduling overhead of exe-
Table 6.1: MAC protocol description and the associated MAC function execution durations measured from WARP board.

<table>
<thead>
<tr>
<th>MAC Protocol Description</th>
<th>Duration [us]</th>
</tr>
</thead>
<tbody>
<tr>
<td>label Start;</td>
<td></td>
</tr>
<tr>
<td>WriteToTxBuffer(DATA);</td>
<td>29.5</td>
</tr>
<tr>
<td>label TryToSend;</td>
<td></td>
</tr>
<tr>
<td>BackOff();</td>
<td>18</td>
</tr>
<tr>
<td>SetFrequencyChannel();</td>
<td>22</td>
</tr>
<tr>
<td>if(CarrierSensing())</td>
<td>30</td>
</tr>
<tr>
<td>SendPacket(DATA);</td>
<td>433</td>
</tr>
<tr>
<td>if(WaitForPkt(ACK))</td>
<td>144</td>
</tr>
<tr>
<td>ReadFromRxBuffer();</td>
<td>20</td>
</tr>
<tr>
<td>goto Start;</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>goto TryToSend;</td>
<td></td>
</tr>
<tr>
<td>endif</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td></td>
</tr>
<tr>
<td>if(WaitForPkt(DATA))</td>
<td>433</td>
</tr>
<tr>
<td>ReadFromRxBuffer();</td>
<td>50</td>
</tr>
<tr>
<td>WriteToTxBuffer(ACK);</td>
<td>11</td>
</tr>
<tr>
<td>SendPacket(ACK);</td>
<td>144</td>
</tr>
<tr>
<td>endif</td>
<td></td>
</tr>
<tr>
<td>goto TryToSend;</td>
<td></td>
</tr>
<tr>
<td>endif</td>
<td></td>
</tr>
</tbody>
</table>

cutting a MAC protocol. It can be seen that the overheads of using two PEs and three PEs are almost the same while the one PE results in a significant increase in the scheduling overhead. It is mainly due to the shortened total execution time of the program by being able to parallelize MAC processes onto multiple PEs. The initialization overhead is around 3% for one iteration of the MAC protocol. Since MAC protocol runs a long time, the initialization overhead is almost negligible. There is no significant difference among the initialization overhead in this test case since the number of PEs used is small. The initialization overhead of the whole P2012 fabric varies significantly with different number of clusters and PEs as we are presenting in Section 6.3.2.

6.3.2 Genetic Algorithm for MAC Parameter Optimization

Genetic Algorithms (GAs) have been widely used in the MAC research community for parameter optimization and performance adaptations. GAs [195] are a method of search, mainly for learning and optimization purposes. Although GA is typically not fast to converge, it is robust, scalable and well sui-
Figure 6.6: The ratio of the execution time of the MAC protocol described in Table 6.1 using multiple PEs for parallelization against sequential execution on one PE.

Figure 6.7: The execution overhead of the MAC protocol described in Table 6.1 on different numbers of PEs in one cluster with different numbers of iterations.
6.3. MAC PARALLELIZATION ON MANY-CORE P2012 PLATFORM

...ted for optimization problems involving large search spaces. The GA computation starts from definition of fitness functions which are measures of performance towards an or multiple objectives. A few randomly generated populations of individuals known as chromosomes are selected. Each chromosome represents a possible solution to a problem. The fitness of each chromosome in a population in each generation is evaluated based on the fitness functions. Crossover and mutation are actions performed to the chromosomes of the populations at each generation. In general, chromosomes which are believed to be able to survive from generation to generation are selected and the population of the new generation is expected to be better than the old generation. The process is iterated until convergence criteria are met [196].

GA has been used for an engine which provides awareness-processing, decision-making and learning elements of cognitive functionality [197, 198]. Since CRs are required to adapt based on the environmental sensing and learning results, GAs are used to evolve a radio defined by a chromosome. The adjustable parameters of a radio are presented by genes in a chromosome. The radio is optimized according to the defined fitness function and the optimal values of the parameters for the CR node is found by GA. GA has also been used in other aspects for CRs such as spectrum management [195], determining of Radio Frequency (RF) parameters for optimal radio communications in the varying RF environment for autonomous vehicle communications [199], and channel allocation [200]. Although GA has been proposed to be used in various areas for CRN, the high computational requirement of the algorithm has been a hurdle to realize GA in real-time for resource optimization in the MAC layer [201]. In order to improve the convergence time and the optimization results of spectrum utilization based on GA, Chen et al. have presented several optimization techniques such as population adaptation, variable adaptation, and manipulation of constraints and objects [202].

Since it is an inherent nature of GAs that the evaluation of individuals can be conducted independently, the computing time can be effectively accelerated by means of parallel computation [203]. Parallel GAs also have the advantage of modelling natural evolution more closely by introducing the concept of spatial locality [204]. Therefore, we see great potential in employing GA for MAC layer optimization at runtime by exploiting the parallelism on many-core architectures.

Problem Formulation

In this work, we implement a GA with multi-objectives. Our implementation is adapted based on previous work from Newman et al. [201]. We have included a list of PHY and MAC parameters as the genes for the chromosomes. The genes include radio transmission power, modulation type, modulation index, frequency channel, number of subcarriers, channel coding rate and packet size. A population of 100 chromosomes each of length of 44 bits are generated...
randomly. The crossover rate is set to be 90% and mutation rate is 5%. The fitness function is defined as

\[ f = w_1 \times (f_{\text{min, power}}) + w_2 \times (f_{\text{min, per}}) + w_3 \times (f_{\text{max, throughput}}), \]  

(6.1)

where

\[ f_{\text{min, per}} = 1 - \frac{\log 0.5}{\log P_{pe}}, \]  

(6.2)

and

\[ P_{pe} = 1 - (1 - P_{be})^L. \]  

(6.3)

The definitions of \( f_{\text{min, power}} \), \( f_{\text{max, throughput}} \) and \( P_{be} \) are according to \cite{201} while \( f_{\text{min, per}} \) refers to the minimum packet error rate \( P_{pe} \), which is related to the bit error rate \( P_{be} \) and the length of the packet \( L \). \( w_1 \), \( w_2 \), and \( w_3 \) are the weights assigned to each individual fitness functions depending on the application scenario and user requirements.

Since the fitness function execution is computationally intensive, involving complicated mathematical functions, we have parallelized the execution of fitness function calculation across chromosomes. Each chromosome is assigned to a PE for the fitness function calculation while the rest of the processes such as mutation, crossover, chromosome selection and replacement is done at the fabric controller.

**Execution Time**

In this experiment, we execute the GA using varying number of clusters and PEs to analyze the benefit and overhead in terms of execution time. Figure 6.8 shows the overhead of using multiple clusters and PEs on P2012 for executing the GA. The initialization overhead increases significantly with the number of clusters while the termination overhead is comparatively negligible. These overheads just occurs once, i.e. when a node is booted and shut-down. Therefore, they are insignificantly small (less than 80 ms) when the lifetime of a node can be of days and months.

Figure 6.9 shows the actual execution time (excluding the initialization and closing overhead) of the GA on P2012 fabric. We can see that with small number of generations, the execution time does not differ too much with different amount of computational power, especially with multiple clusters. However, a GA typically reach a fitness score of 0.8 and above at approximately 200 iterations. Significant improvement is shown when using more clusters and more PEs as the number of generation increases. In an extreme case, an 85% improvement is observed for 200 iterations when four clusters with 64 PEs in total are used as compared to one PE. It shows that when a change has occurred in terms of either application QoS requirements, or special condition, or network topology, etc., which demands an adaptation at the MAC scheme, the node is able to find an acceptable solution in 10 ms if equipped with the whole P2012 fabric for parallel execution as compared to 75 ms with sequential execution.
Figure 6.8: The initialization and termination overhead of using different number of PEs and clusters of the P2012 fabric when executing the parallel GA.

Figure 6.9: The execution time of the GA with different number of generations using different number of PEs and clusters of the P2012 fabric.
6.3.3  **Swarm Intelligence Algorithm for Channel Allocation**

Inspired by the observation that social insects such as ants and bees work in a self-organized fashion with unsupervised coordination between simple interactions among individuals in the colony, Swarm Intelligence (SI) algorithms model network users as a population of simple agents interacting with the surrounding environment \[8, 205\]. Although each agent has little intelligence, global intelligence is resulted from the collaborative behaviour of the colony. Division of labour is the key in SI where different activities are performed by those who are better suited to the task. SI algorithms have three characteristics which made them popular in a wide range of applications: flexibility in adapting to a changing environment, robustness against failure of individuals, and self-organization with unsupervised activities \[206\]. Although in this work we have applied SI for MAC layer channel allocation, SI has been widely popular in resource management in CRNs in general. Doerr et al. have used SI algorithm to dynamically identify common control channel in CRNs \[207\]. SI algorithms have also been used for optimum resource allocation in terms of assigning available spectrum holes to CR users \[208, 209\]. BIOlogically-inspired Spectrum Sharing (BI OSS) algorithm allocates channels to unlicensed users in CRNs based on the adaptive task allocation model in insect colonies. BIOSS works with distributed network architecture since the users can distributively select the channels for communication \[210\]. BIOSS has been enhanced in channel allocation so that channels which have the minimum excess power over the node’s transmission power is preferred \[211\]. This modification has improved the utilization of low-power channels and thus the global spectrum utilization.

**Problem Formulation**

We have followed the enhanced BIOSS protocol (eBIOSS) \[211\] closely for our channel allocation algorithm implementation. A brief overview of the algorithm is described here. The probability of selecting a particular channel which satisfies the transmission requirement is

\[
T_{ij}^{\text{esp}} = \begin{cases} 
0, & P_j < p_i \\
1 - \frac{P_j^{n}}{P_j^{n} + \alpha p_{ij}}, & P_j \geq p_i 
\end{cases}
\]  

(6.4)

where \(P_j\) is the permissible power to channel \(j\), \(p_{ij}\) is the required transmission power of node \(i\) to channel \(j\). It is assumed that the permissible power to all channels are available at each node through spectrum sensing, and the required transmission power can be determined according to the user requirements and channel characteristics. \(n\) determines the slope of the channel selection probability \(T_{ij}^{\text{esp}}\) and is set to be 2 in our implementation. \(\alpha\) is a constant which determines the influence of \(p_{ij}\) and is set to be 10.
The channel selection probability is calculated for all the channels at a node when an event happens such as a transmission task arises, the operating environment changes, the QoS requirement varies, etc. The channel with the maximum $T_{ij}^{cap}$ is selected. The selection is then evaluated and good channels are remembered.

### Channel Allocation

In our experiment, we assume a one-hop network with fixed number of channels and nodes contending for transmission. Each node is assigned with a required transmission power per transmission task randomly. Each node is equipped with a P2012 fabric. We initialize each node with a random number of clusters (1 to 4), which helps us to evaluate the relationship between the processing capability and the opportunity in finding a suitable channel for transmission. Since there is typically more demand than supply, i.e. more nodes within a network than the channels which satisfy the required transmission power, some nodes might not find a suitable channel every time. Channel algorithms are performed repetitively until a channel is allocated for the pending transmission task. Figure 6.10 shows the possibility for a node to find a

![Figure 6.10: The channel allocation rate of varying sizes of network with different number of clusters for executing the channel allocation algorithm. The total number of channels is 20.](image)
suitable channel in a network of varying size. We see that the nodes which only have one cluster for executing eBIOSS algorithm performs the worst while the rest nodes have similar level of performance. It is due to the fact that since the number of total available channel is 20 and the number of PEs in one cluster is only 16, it takes much longer for the nodes with one cluster to calculate the channel selection probability for all the channels. Since channels are allocated based on a first come first serve basis, the nodes which are able to process information faster have higher chance in grabbing the channels which fit to their transmission requirement.

Figure 6.11 shows the possibility to obtain a suitable channel in a network with different number of channels available. The more channels available, the more computational power is required from the node to be able to evaluate the channel quality on time. We see that when only 10 channels are available, nodes with more clusters do not have any advantage over nodes with only one cluster and the successful channel allocation rates are the same for all the nodes. When the number of channels increases, high computational power becomes more beneficial.

![Figure 6.11](image_url)

Figure 6.11: The channel allocation rate at nodes with different computational power of a network with 30 nodes. The number of channels varies from 10 to 60.
6.4 SUMMARY AND DISCUSSIONS

Multi-core architectures are nowadays common in general purpose computing and in high performance computing. For wireless communication networks, multi-core platforms are becoming a promising alternative for reducing single core design complexity and power consumption [212]. The trend of having multiple transceivers and very fast baseband systems on silicon is also imposing requirements for MACs and particularly soft-MACs to such level that multi-core implementation are becoming requirement. With increasing requirements of hard real-time response and high degree of flexibility for MAC-layer algorithms, SDR community is exploring both heterogeneous as well as homogenous many-core architectures. Therefore, we explore the possible performance improvement through MAC protocol parallelization on multi-core/many-core platforms using TRUMP.

We have discussed on the various platform architectures for software defined radios and their advantages and disadvantages in terms of flexibility, programmability, area and power efficiency, and the ability to meet the timing requirement of PHY/MAC layer processes. Since heterogeneous platforms exhibit restricted possibilities for extension and legacy code compatibility, we investigate parallelization in MAC processes for efficient realization on the P2012 many-core computing fabric with homogeneous processing elements. We have implemented TRUMP on the target platform to exploit the parallelism and schedules different MAC processes on different computing elements according to the state-machine of a particular MAC scheme while fulfilling the real-time constraints. We have considered different classes of MAC scheme realizations. We have observed that classical simple CSMA/CA based MAC protocols result in limited benefit from parallelization. This is owing to the fact that these protocols do not involve heavy computations and the MAC processes exhibit short execution and blocking times. MAC protocols requiring advanced channel selection and resource management schemes, mainly using machine learning and statistical data analysis methods, have a high potential for exploiting the computational power and parallelism. With high computational demands, tight scheduling and timeliness constraints of the MAC schemes, these algorithms cannot be offloaded externally due to data and control bottlenecks. We have shown that by fully exploiting the computational power on P2012, we are able to achieve an up to 85% improvement in convergence time when using genetic algorithm for MAC/PHY parameter optimization. We have also shown when using SI algorithm for channel allocation, it is 2-6 times more likely for a node with more computing power to get a desirable channel than a node with limited computational power. We believe that our results show the importance of parallelization of computationally complex MAC strategies and emphasize the need for many-core architecture in SDR platforms.
Traditionally, MAC protocols are implemented in a static and monolithic way as they are designed for a particular platform to support a specific application, thus allowing limited possibility for reconfiguration and adaptation. However, wireless networks these days are expected to support multiple applications and operational modes. Therefore, the complexity of MAC protocols keeps on increasing. The medium access procedure is no longer just simply transmitting at a given time slot or a given channel. Due to the unstableness of the spectrum environment for wireless network, wireless terminals are required to dynamically identify spectrum holes in a wide band, select the available channel while not causing interference to other co-existing networks. Static MAC implementation approach no longer suffices the current situation and requirements. Therefore, adding flexibility to MAC protocols to allow reconfiguration and enable MAC adaptation is the way forward in MAC protocol development for wireless networks. Furthermore, as the number of platforms available for wireless nodes increases, MAC code cross-platform portability and re-usability is also a key issue in protocol design. In this dissertation, we have proposed Decomposable MAC Framework for rapid MAC protocol prototyping, which has been shown to be easily portable among platforms. The framework, together with our designed toolchain, enabled flexible MAC realization and runtime reconfiguration. We have implemented the framework and toolchain on wireless sensor nodes and WARP SDR development boards to show the wide applicability of our design. Using the toolchain based on the framework, we have enabled different ways of runtime optimization for MAC protocols. We have also shown MAC performance enhancements by exploiting multi-core platform architecture which we have identified to be the trend for wireless terminal development for the new generation of wireless networks. In this chapter, we summarize our results in Section 7.1 and identify some future work directions in Section 7.2.

7.1 Summary of Results

In this dissertation, we have presented a comprehensive study of the state-of-the-art efforts in flexible MAC protocols research to support the increasing demand from wireless applications and the increasing complexity of network conditions. We see a need for a user-friendly tool for easy and fast MAC pro-
protocol prototyping and deployment onto hardware platforms. Therefore, we have designed Decomposable MAC Framework, a flexible MAC development framework based on the MAC decomposition philosophy, as the foundation work to fulfil the need. Our framework enables high level of code re-usage, easy portability among different platforms, and provides means for achieving rapid MAC protocol design and prototyping. We have identified a list of common MAC components which serve as the basic building blocks for a vast range of MAC protocols. Using these blocks, we have implemented a large set of MAC protocols including IEEE 802.11 MAC like protocol with advanced features such as data aggregation and block acknowledgements; and spectrum agile MAC protocol CogMAC, which is designed for CRNs and suitable for infrastructureless wireless networks. These implementations show that Decomposable MAC Framework can be used to realize hugely different and complex MAC protocols in a rapid fashion. We have realized Decomposable MAC Framework on both WARP SDR boards and TinyOS compliant sensor nodes. A high level of component and code reuse has been observed for different MAC realizations. We have presented MAC-PD, which is designed based on Decomposable MAC Framework. It provides a user-friendly graphical user interface where MAC protocols can be realized by “drag-drop-and-connect” components together. The designer can design MAC protocols in the form of flow-charts. Code is automatically generated and deployed onto target platforms.

MAC schemes require a high degree of flexibility and runtime adaptability in order to cope with the emerging challenges such as to satisfy varying application demands, user patterns, changing network conditions, and to efficiently utilize the spectral resources. Decomposable MAC Framework serves as a foundation work in this dissertation to introduce flexibility to MAC protocols. The component-based approach provides the opportunity of tool development for enabling MAC reconfiguration with a high level of flexibility and low level of implementation overhead. Using Decomposable MAC Framework, we have implemented TRUMP, a toolchain which enables runtime MAC protocol compositions by binding pre-defined reusable protocol components together. Our design provides fast reconfiguration speed and a lightweight implementation in order to meet time-critical requirements imposed by MAC processing. TRUMP provides a rich set of tools for developing complex MAC schemes including a MAC meta-language descriptor, a MAC meta-compiler and Wiring Engine. A “drag-and-drop” based GUI in the similar fashion as MAC-PD is also provided to enhance user experience using flow-charts for designing MAC protocols. Our evaluation results show that TRUMP implementation for WARP boards enables fast runtime composition of protocols ranging from a simple CSMA based MAC to a multichannel spectrum agile MAC protocol. Furthermore, we have adapted the design of TRUMP to typically resource constrained platforms such as sensor nodes. The results show that our implementation well fits the memory limit of the sensor nodes.
7.1. **Summary of Results**

and the execution time overhead using the toolchain is only 1% for the widely used preamble sampling and common active period MAC protocols. Our approach allows realizations of a vast list of MAC protocols at runtime using the given limited memory resource.

TRUMP also provides the capability of MAC protocol performance optimization by offering easy reconfiguration of MAC protocols. Since a wide range of protocols can be realized at runtime, TRUMP allows the current protocol to be adapted according to any changes in spectral environments, user preferences, application requirements, etc. We have introduced a runtime MAC optimizer which uses TRUMP for MAC adaptation. In order to optimize the performance at the MAC layer, we monitor the network environment, application parameters and runtime MAC performance feedback at the same time. Our experimental results show that with a runtime MAC optimizer, we have enabled flexible MAC adaptation both in terms of parameter tuning and functional re-composition. Our MAC protocol is able to provide a performance gain in terms of throughput up to 400% in a severely interfered environment in our test case. In order to enable runtime protocol realization tailored to a wide range of user-specified application preferences, we benefit from a MAC meta-compiler which is part of TRUMP. The application can impose multiple criteria for a MAC realization such as maximizing energy efficiency, minimizing latency and maximizing data reliability through the MAC meta-compiler at runtime. The meta-compiler interacts with the runtime optimizer to select the most appropriate MAC parameter and functions according to the optimization criteria. We have demonstrated different MAC adaptation behaviours under different application preferences. To ensure that the node specific local optimization of MAC protocol does not bring detrimental effect to the overall network performance, such as loss of communication between nodes due to extremely different spectral conditions they experienced individually, we have implemented a cooperative scheme which is integrated with TRUMP. The cooperative scheme keeps track of the MAC states of all the neighbouring nodes and triggers necessary MAC state exchange when a MAC adaptation is to be performed at runtime. Our results show that a seamless communication has been achieved within a highly dynamic spectral environment in a multi-hop network where the cooperative scheme handles control information exchanges.

With increasing requirements of hard real-time response and high degree of flexibility for MAC layer algorithms, SDR community is exploring both heterogeneous as well as homogenous many-core architectures for mobile terminals. TRUMP provides dependency management among MAC components, which in turn provides the possibility to execute two or more independent components in parallel. It also offers scheduling and mapping of processes onto available hardware resources. Therefore, we explore the possible performance improvement through MAC protocol parallelization on multi-core platforms using TRUMP. We have first experimented with Linux based x86
multi-core architecture. Our results have confirmed the potential of MAC protocol parallelization. Therefore, we continued to investigate in the SDR platform architectures for MAC protocol execution. We have discussed the advantages and disadvantages of various SDR platform architectures in terms of flexibility, programmability, area and power efficiency, and the ability to meet the timing requirement of PHY/MAC layer processes. Since heterogeneous platforms exhibit restricted possibilities for extension and legacy code compatibility, we investigate parallelization in MAC processes for efficient realization on the P2012 many-core computing fabric with homogeneous processing elements. We have implemented TRUMP on the target platform to exploit the parallelism. TRUMP schedules different MAC processes on different computing elements according to the state-machine of a particular MAC scheme while fulfilling the real-time constraints. We have considered different classes of MAC scheme realizations. It has been observed that classical simple CSMA/CA based MAC protocols result in limited benefit from parallelization. This is owing to the fact that these protocols do not involve heavy computations and the MAC processes exhibit short execution and blocking times. MAC protocols requiring advanced channel selection and resource management schemes, mainly using machine learning and statistical data analysis methods, have a high potential for exploiting the computational power and parallelism. With high computational demands, tight scheduling and timeliness constraints of the MAC schemes, these algorithms cannot be offloaded externally due to data and control bottlenecks. We have shown that by fully exploiting the computational power on P2012, we are able to achieve up to 85% improvement in convergence time of the genetic algorithm used for MAC/PHY parameter optimization. We have also shown when using SI algorithm for channel allocation, it is 2-6 times more likely for a node with more computing power to get a desirable channel than a node with limited computational power. We believe that our results show the importance of parallelization of computationally complex MAC strategies and emphasize the need for many-core architecture in SDR platforms.

7.2 Future Work

In this dissertation, we present a tool and framework for reconfigurable MAC protocol realization. There are multiple enhancements and research directions that are possible based on our presented work.

So far, the experiments we have carried out for evaluation of our reconfigurable MAC approach and the related runtime optimizations are using a small number of nodes to form one- or two-hop networks. Although small network can best represent the MAC layer characteristics enabled by our tool, it would be interesting to see a large-scale network wide behaviour. Due to the limited availability of WARP boards, a large-scale network test can be carried out
by using sensor nodes. Needs for MAC/Routing runtime reconfiguration and cross-layer optimization might rise during the experimentation which can be a direction for further development of our toolchain.

For parallelization of MAC processes, we have experimented with multi-threading on an x86 Linux based machine and a cycle accurate many-core fabric emulator. Neither of the test platforms has radio front-end attached which made our investigation at best a good estimate of the possible actual performance. We are currently investigating enabling multi-threading and dual-core support on WARP boards. We have customized Xilkernal, a real-time operating system, for WARP boards [415]. TRUMP implementation can be integrated with the operation system. The Virtex II FPGA on WARP boards has two PowerPC cores which can be utilized to test our parallelized runtime parameter optimization and channel allocation schemes based on machine learning algorithms in a realistic environment.
ABBREVIATIONS

ACK  Acknowledgement
A-MPDU  Aggregate MAC Protocol Data Unit
A-MSDU  Aggregate MAC Service Data Unit
API  Application Programming Interface
ASIC  Application-Specific Integrated Circuit
ASIP  Application-Specific Instruction-set Processor
BA  Block Acknowledgement
BAR  Block Acknowledgement Request
BIOSS  BIOlogically-inspired Spectrum Sharing
BPSK  Binary Phase Shift Keying
CC  Cluster Controller
CCC  Common Control Channel
CCM  CORBA Component Model
COM  Component Object Model
CORBA  Common Object Request Broker Architecture
COTS  Commercial Off-The-Shelf
CPU  Central Processing Unit
CR  Cognitive Radio
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRC</td>
<td>Cyclic Redundancy Check</td>
</tr>
<tr>
<td>CRN</td>
<td>Cognitive Radio Network</td>
</tr>
<tr>
<td>CS</td>
<td>Carrier Sensing</td>
</tr>
<tr>
<td>CSMA</td>
<td>Carrier Sense Multiple Access</td>
</tr>
<tr>
<td>CTS</td>
<td>Clear To Send</td>
</tr>
<tr>
<td>DCF</td>
<td>Distributed Coordination Function</td>
</tr>
<tr>
<td>DFP</td>
<td>Data Frame Preamble</td>
</tr>
<tr>
<td>DFS</td>
<td>Dynamic Frequency Selection</td>
</tr>
<tr>
<td>DIFS</td>
<td>DCF Interframe Space</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DSA</td>
<td>Dynamic Spectrum Access</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>EJB</td>
<td>Enterprise JavaBeans</td>
</tr>
<tr>
<td>EU</td>
<td>European Union</td>
</tr>
<tr>
<td>FCS</td>
<td>Frame Check Sequence</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field-programmable Gate Array</td>
</tr>
<tr>
<td>GA</td>
<td>Genetic Algorithm</td>
</tr>
<tr>
<td>GPP</td>
<td>General Purpose Processor</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>GSM</td>
<td>Global System for Mobile communications</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
</tbody>
</table>
GPIO  General Purpose Input Output
IEEE  Institute of Electrical and Electronics Engineers
ISM   Industrial, Scientific, and Medical
ISR   Interrupt Service Routine
LPL   Low Power Listening
LTE   Long Term Evolution
MAC   Medium Access Control
MAC-PD MAC Protocol Designer
MANET Mobile Ad-Hoc Network
MCCF  Many-Core Computing Fabric
MFP   Micro Frame Preamble
MLA   MAC Layer Architecture
MPDU  MAC Protocol Data Unit
MPSoC Multiprocessor System-on-Chip
MSDU  MAC Service Data Unit
NIC   Network Interface Card
NoC   Network-on-Chip
NPM   Native Programming Model
OFDM  Orthogonal Frequency Division Multiplexing
PE    Processing Element
PHY   Physical (layer)
PSM   Power Saving Model
PU  Primary User
QAM  Quadrature Amplitude Modulation
QoS  Quality of Service
QPSK  Quadrature Phase Shift Keying
RAH-MAC  Rate Adaptive Hybrid MAC Protocol
RAM  Random-Access Memory
RBAR  Receiver-Based AutoRate Protocol
RF  Radio Frequency
RNG  Random Number Generator
ROM  Read-Only Memory
RSSI  Received Signal Strength Indicator
RTM  Reactive Task Manager
RTS  Request To Send
SDR  Software Defined Radio
SIFS  Short Interframe Space
SRAC  Single-Radio Adaptive Channel
SU  Secondary User
TCDM  Tightly Coupled Data Memory
TCP  Transport Control Protocol
TDMA  Time Division Multiple Access
TPC  Transmit Power Control
TrawMAC  Traffic Aware Medium Access Control
TRUMP  Toolchain for RUniMe Protocol realization
ULLA  Unified Link-Layer API
UPMA  Unified Power Management Architecture
USRP  Universal Software Radio Peripheral
VANET  Vehicular Ad-Hoc Newtork
WARP  Wireless Open-Access Research Platform
WLAN  Wireless Local Area Network
WRAN  Wireless Regional Area Network
WSN  Wireless Sensor Network


CURRICULUM VITAE

Personal Information

Name Xi Zhang
Date of birth 11 February 1983
Place of birth Shanxi, P.R.China
Nationality Chinese

Education

09.2007 – 07.2014 Doktor der Ingenieurwissenschaften Communications Engineering
RWTH Aachen University Germany

10.2003 – 08.2007 Master of Engineering Electrical and Electronic Engineering
Imperial College London United Kingdom

Experience

11.2013 – current Research Scientist
Corporate Technology, Siemens AG Germany

09.2007 – 09.2013 Research Assistant
iNETS, RWTH Aachen University Germany

07.2006 – 08.2006 Summer Intern - Risk Management Analyst
China Minsheng Banking Corporation China

07.2005 – 09.2005 Summer Intern - Corporate Auditor
Siongbee Packaging (Wuxi) co. Ltd. China

07.2004 – 08.2004 Summer Intern - Project Engineer Assistant
Huntsman Polyurethane Shanghai China
LIST OF PUBLICATIONS

JOURNAL ARTICLES RELEVANT TO THE THESIS TOPIC:


CONFERENCE PAPERS RELEVANT TO THE THESIS TOPIC:


ARTICLES ADDITIONAL TO THE THESIS TOPIC:

