A Compiler Assisted Approach for Component based Reconfigurable MAC Design

Junaid Ansari, Xi Zhang and Petri Mähönen
Institute for Networked Systems, RWTH Aachen University, Kackertstrasse 9, D-52072, Aachen, Germany
Email: {jan, xzh, pma}@inets.rwth-aachen.de

***Abstract*—Cognitive radio networks require reconfiguration and adaptivity in order to efficiently meet the changing application demands and network conditions. We have developed a framework which allows composition of MAC protocols using a library of MAC components [1]. These components are implemented with a hardware-software co-design approach so as to satisfy the timeliness requirements as well as to provide the desired degree of flexibility. A domain specific MAC language and corresponding MAC-meta compiler toolchain is developed to realize highly dynamic and reconfigurable MAC solutions using the MAC components. The prototype implementation on WARP [2] SDR boards indicates that our approach eases the MAC development without compromising on the performance characteristics as compared to the monolithic way of implementing MAC protocols.**

I. Introduction

Cognitive radio networks are expected to cope with changing application requirements, adapt to network characteristics and observe spectrum policies. They are becoming the technological foundation for efficiently managing the scarcity of wireless spectrum, fulfilling various QoS demands and allowing different networks to coexist. Cognition and spectrum agility in MAC protocols require adaptability and close PHY-MAC interaction. Apart from the needs of cognitive radio technologies there are many reasons to consider reconfigurable MACs and rapid development design tools. As an example, current development of many innovative MAC concepts particularly towards IEEE 802.11 extensions are severely hindered by the fact that it is difficult to implement MAC protocols and test them in realistic environments. One of the key reasons is that MAC protocols have been classically implemented in hardware, which gives a limited possibility for reconfiguration and customization. Recently, software based MAC implementations have emerged although a close hardware-software co-design is typically required to keep time critical operations in silicon.

We introduce a MAC development framework for enabling fast composition of MAC protocols. The framework is developed to provide tools for rapidly developing MACs that are best fitted to the application requirements, communication capabilities of the radios, and spectrum regulations [1]. We have analyzed a wide range of MAC protocols and identified their basic common functionalities. These functionalities are provided as "building blocks" to compose different protocols. Our framework provides a toolchain which allows on-the-fly realization of an envisioned MAC protocol through wiring of these common MAC components. We have developed a MAC meta-language, which expresses the state-machine logic, dependencies among the components and the execution flow. This enables fine-grained control over the binding logic of the components and allows expressing an entire MAC protocol with just a few lines of code. Correspondingly, a meta-compiler tool-chain running on the host node realizes the MAC protocol in an autonomous way. Since a particular MAC is realized through simply wiring and executing the components according to the state-machine logic, this approach easily allows modification in the MAC composition at run-time and hence enables rapid reconfiguration.

The rest of the paper is structured as follows. Section II discusses the related work towards flexible and reconfigurable MAC development. Section III discusses in detail the design rationale of the various components of our framework while Section IV describes the implementation details on WARP boards. In Section V, we present the experimental evaluation results and finally conclude the paper in Section VI.

II. Related Work

MAC protocols are typically implemented in a monolithic fashion with tight coupling to the underlying hardware. A hardware specific implementation restricts the reconfigurability and flexibility aspects. Furthermore, it limits the portability and design extension possibilities. Efforts have however been made to achieve a unified protocol structure for easing implementation burden and maximizing code reuse. Polastre *et al.* proposed a unified link layer abstraction, which helps the implementation of a broad range of networking and data link technologies without significant loss in efficiency [3]. ULLA [4] focuses on the design of flexible link layer APIs. It offers a common interface to fetch link layer information independent of the underlying radio technology. Bianchi *et al.* discuss the advantages of an adaptive and programmable MAC framework, which satisfies time varying QoS demands [5]. MultiMAC [6] allows switching among a few pre-defined standalone MAC solutions to suit the changing network and application requirements. This approach has a limited scope because it can only find an approximate "best-fit" solution from a limited number of protocols in the MAC pool.

Messerschmidt has discussed in detail the modularity and component oriented design approach [7]. Braden *et al.* [8] have proposed role based approach using the component oriented and non-classical view for protocol designing to allow

978-1-4577-0899-2/11/$26.00 ©2011 IEEE

reusability, flexibility and portability. Unfortunately, the protocol heap approach lacks actual implementation. In order to provide reconfigurability features to a MAC developer through flexible interfaces, Sharma *et al.* propose FreeMAC [9], which is a multichannel MAC development framework on top of the standard IEEE 802.11 hardware. FreeMAC aims at supporting frequent channel switching and efficiently controlling the timings for packet transmissions. A similar work has been demonstrated in [10]. In [11], the authors control the timings of the frames through the MadWifi driver for Atheros based NICs in order to improve the overall throughput. In [12], authors show a TDMA MAC protocol implementation in software (Linux user-space) using the MadWifi driver for Atheros AR5212 chipset. Han *et al.* have shown significant packet recovery improvements by using a negative acknowledgement scheme for block-wise checksums and a fine grained control over the Broadcom wireless NICs using OpenFWWF [13]. However, since IEEE 802.11 hardware is restricted in providing accessibility to the radio and PHY parameters as are needed by many spectrum agile and cognitive MACs, the configurability aspects of these protocols are limited. Furthermore, these designs are not implemented in a modular fashion to allow portability – only a few MAC parameters can be changed.

In recent years, software based MAC implementations using for instance GNU radio [14], etc., have emerged to offer higher radio and physical layer reconfigurability but these have shortcomings in meeting time critical deadlines. SDR platforms have demonstrated possible advantages in the MAC designs based on the flexible radios. Some of the low-cost SDR-platforms do not offer real-time performance characteristics and flexibility as has been learnt by other research groups in the case of USRP [15] and GNU Radio platforms [16], [17]. SORA [18] has been demonstrated with IEEE 802.11 a/b/g software implementations on a standalone PC. WARP [2] provides experimental opportunities for customizing the hardware and prototyping MACs working in autonomous fashion. In [19] authors have implemented a fully distributed cooperative PHY/MAC system on WARP boards with accurate timing characteristics. However, despite the fine-grained access control knobs offered by the platform, current MAC development frameworks on SDR platforms lack the support for flexible MAC design and fast on-the-fly reconfiguration.

In order to realize efficient MAC implementations, Nychis *et al.* [17] have split the MAC functionalities based on the timing characteristics and their latency tolerances. Following the similar design methodology, we have proposed Decomposable MAC Framework [1], which partitions the PHY/MAC component implementation in software and hardware in a way that time critical components reside in hardware for computing and communication speed gains while flexible binding of the PHY/MAC components and the MAC state-machine logic is implemented in software. The concept of component interconnections has long been established in software engineering since a software waveform can be expressed as a network of boxes communicating with each other via connecting buses. Compositional adaptation techniques have been well investigated [20] because of the need for run-time reconfigurations, such as in robotic software systems. Networking researchers have actively used Click [21] for binding components together however in order to allow fast speed and suit to limited resources, a customized tool is more suitable as we show later in our evaluation. Unlike the approach of Nychis *et al.*, in the Decomposable MAC Framework [1], we have defined the PHY/MAC components not just based on the timeliness requirements but also communication demands and the degree of reuse. Compared to [6], our approach does not require large memory for storing the perceived protocols to be able to switch among them as the application requirements evolve. We have followed this approach in the German Research Foundation (Deutsche Forschungsgemeinschaft) funded Nucleus [22] and EU funded 2PARMA projects [23]. Realizing components with high communication and computational demands on multi-core platform offers a new perspective for dynamic protocol realization. In order to benefit from this approach and exploit the platform architecture, we have also proposed a supporting toolchain known as TRUMP: Toolchain for RUntiMe Protocol realization [24]. A compiler assisted approach for developing MAC protocols as discussed in [25] provides a systematic way of automation of MAC implementation, analysis and code generation. However, it is specifically designed for co-operative MAC solutions, therefore fails to address the wide scope of MAC realizations. Furthermore, timeliness demands and real-time constraints are not considered by the approach.

III. Framework Design

In order to facilitate fast MAC implementation and reconfigurable designs, we have developed a framework, which realizes MAC protocols based on the common functional components through a rich set of tools as a part of the framework. Our toolchain consists of

1) a Wiring Engine which binds the MAC components,
2) a meta-language to describe the MAC design,
3) a (host) compiler which converts the MAC language to executable code for a particular target platform,
4) an interactive Graphical User Interface (GUI) to ease the MAC designing.

Fig. 1 shows the MAC realization process in our framework. The 'Drag and Drop' based interactive GUI provides an IDE for the developers to design a MAC in the form of flowcharts, which is converted into the domain specific language and is sent to the target platform over a serial bus interface. The host compiler parses the code and maps it to the MAC components. The Wiring Engine controls the state-machine of the MAC (i.e., coordinates the control and data flow) and executes the MAC components. In the following, we will describe the various parts of the toolchain.

A. Component Oriented MAC Design Approach

MAC protocols show many functional commonalities. We have analyzed different protocols based on CSMA/CA, TDMA

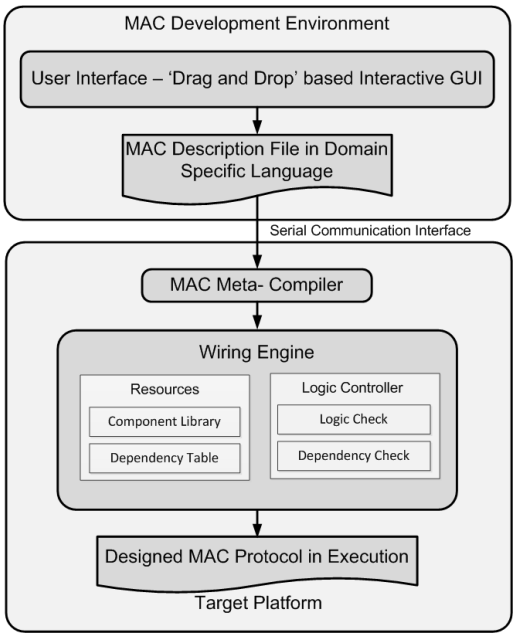

Fig. 1. An illustration of the MAC realization process.

and hybrid principles in order to identify the common functionalities so that different protocols can be implemented based on the same set of functional components or "building blocks". MAC functionalities such as timer, backoff counter, carrier sensing, frame formation, sending a frame, receiving a frame, etc., are not only common to different protocols but may also be repeated within a particular protocol realization. Additionally, access to radio control parameters like switching the state of the radio (transmit, receive, sleep), setting the transmit power level, tuning the receiver sensitivity value, selecting the operating frequency, etc., are also typically needed by different MAC implementations. We have designed flexible APIs for each of these components so that those may be reused across a wide range of protocols.

The pattern in which various components are connected is identical across different MAC implementations. Based on this observation, we have defined a set of secondary level MAC components, which are composed of basic components and/or other secondary components. Table I lists the most commonly used secondary level components. Making use of secondary components further simplifies MAC design since they can capture much complex functionalities.

B. Wiring Engine

The approach of binding components through the Wiring Engine widens the possibilities for MAC realization. The wiring approach also enables dynamic adaptation of MAC behaviors to the changing environmental conditions and application requirements. The concept of wiring has been long established in component based systems since a software waveform can be expressed as a network of boxes communicating with each other via connecting buses. Compositional adaptation

TABLE I
COMMONLY USED SECONDARY LEVEL MAC COMPONENTS.

Component	Usage and the *composition*
Random Backoff	Random backoff mechanism *Timer, Random Number Generator, Carrier Sensing*
Expecting Frame	Used when the node is waiting in anticipation of a packet *ReceiveFrame, Timer, Radio Switching, SendFrame*
Send Packet	Called after seizing a channel free *SendFrame, Expecting Frame, Radio Switching, Random backoff*
RTS/CTS/DATA/ACK	Four-way handshake mechanism *Send Packet, Expecting Frame*

techniques have been well investigated [20], e.g. in the context of run-time reconfigurations in robotic software systems. We have developed a light-weight Wiring Engine which is able to dynamically redirect the construction and execution path of a MAC protocol state-machine. This approach enables both runtime reconfiguration of MAC protocols according to pre-defined rules within the framework, and on-the-fly realization of user configured protocols. The Wiring Engine can be divided into two parts. a) The Wiring Mechanism, which governs the linking of components and b) the Logic Controller, which acts as the brain of the Wiring Engine and performs error checking. The Wiring Mechanism consists of four main parts: 1) component interfaces which wrap the fundamental components of the component library elements with a standard API: `int Function(*radio)`, where pointer to the radio object contains all the radio and control parameters for the component; 2) a linked list to realize component re-/linking at run-time; 3) an instruction set such as IF, ELSE, GOTO, etc, to realize logical and non-sequential operations where these instructions are represented as the elements in the linked list; and 4) a run-time execution manager. In a simple case, running an application in the form of linked list is done by traversing the whole list from head to tail. However, depending on the state-machine logic, the execution manager determines the position of the pointer in the list based on the content of the logical elements.

Instead of merely binding the elements, we take care of the dependencies among elements and the binding logic by using the Logic Controller. The Logic Controller ensures smooth execution of the composed protocol by identifying conflicts between the relationship among components and the execution order. The Wiring Mechanism treats all functional components as independent entities. The sequence of execution of components is managed by the run-time execution manager based on the logical connections. However, in network protocols, some functional components are dependent on each other. In order to reduce design efforts and offer designer the reassurance that only correctly designed protocol can be synthesized and deployed onto the target platform, the Logic Controller reports erroneous MAC designs and governs the integrity of the executed protocols. We have devised pre-defined rules to express interdependencies among components and incorporate them as part of the functional libraries. Based on the dependency table,

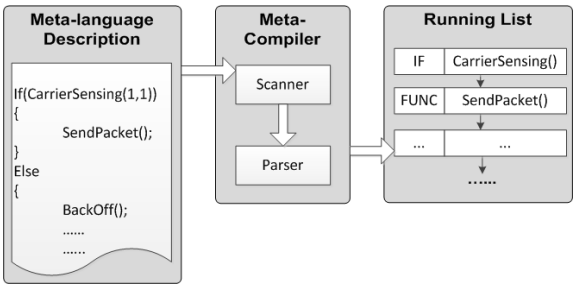

Fig. 2. The MAC compilation process.

our framework explores the opportunity for parallel execution of functions which has a great potential to benefit from multi-core platform architecture as described in Section III-D.

C. MAC Language and Meta Compiler

Based on the component library, we have designed a meta-language descriptor and the corresponding compiler specifically for MAC protocol implementation. Our meta-language has a simple C-like language syntax and a grammar consisting of three basic categories of keywords. This compiler assisted approach prevents potential mistakes in protocol implementation and reduces the protocol development time. The supporting compiler is designed using Lex&Yacc [26] and is integrated to be part of the host node toolchain. The protocol development process using our MAC language and compiler is shown in Fig. 2, where the MAC compiler extracts the protocol logic from meta-description and maps it to functions in the Running List.

1) Identification of Keywords: With comprehensive analysis and understanding of MAC protocols from various classes of wireless networks, we have identified three main requirements in MAC protocol execution: variable declaration, conditional branch and loop. Specifically, we define six keywords: VAR for variable declaration; IF, ELSE, ENDIF for conditional branching and LABEL, GOTO for describing loops. In order to simplify the compilation process, all the values assigned to a variable are declared as integers with associated decimal places. For example, `VAR a = (5,0)` means that `a` is of value 5 while `VAR a = (55,1)` means that `a` is of value 5.5.

2) Compiler Architecture: 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 the 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 accordingly creates

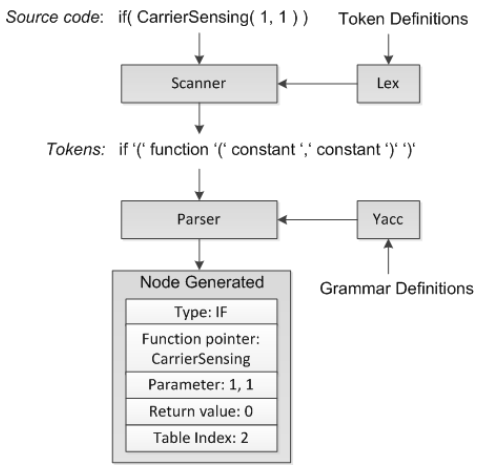

Fig. 3. An illustration of node generation through compiler processes.

nodes for the linked list as shown in Fig. 3. The variables declared in the MAC description file are stored as memory blocks in a static memory pool in a way similar to the heaps of general compiler. Fig. 4 illustrates the variable and constant memory management used by the compiler. The constants used as parameters of functions are stored in a pre-set constant pool.

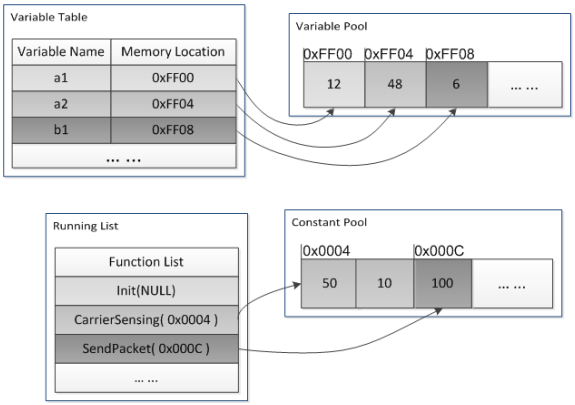

Fig. 4. An example of the memory management for a variable and constant assignment by the MAC meta-compiler.

D. Parallelization and Multi-core Support

Multi-core architectures are nowadays common in general purpose as well as in embedded computing. Many digital signal processor and microcontroller architectures also support multi-core capability. For wireless communication networks, multi-core platforms are becoming a promising alternative for reducing single core design complexity and power consumption. Since parallelism requirements have been identified for network protocol execution [27], we explore the possible performance improvement we can benefit from multi-core platforms. As mentioned in Section III-B, our framework maintains the dependencies among different MAC processes

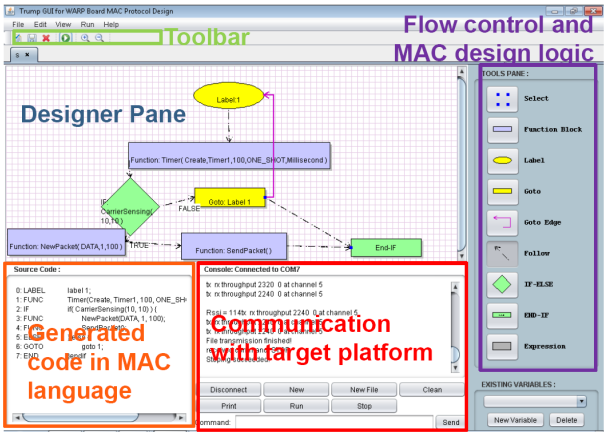

Fig. 5. Snapshot of the MAC designing tool.

and thus enables auto-parallelization of MAC components execution. In addition, we have included a thread pool and a scheduler functionality in our framework so that we are able to schedule different processes onto different processing units. The initial evaluation results are shown in Section V.

E. Drag and Drop based Graphical User Interface

In order to enhance the usability, we have developed and provided a flexible and user-friendly interface for MAC design. It is based on the Java-Swing GUI (c.f. Fig. 5) and allows interactive 'drag-and-drop' feature for MAC development. All the basic and custom defined MAC components are made available to the user. A designer can simply select a component, drag it to the provided drawing grid and set the individual component parameters. The User Interface layer gathers the information of the state-machine and logic flow of the MAC from the user including arithmetic, boolean and logical expressions. A user sets the states of the nodes through dialog screens and defines connections among the nodes to complete the MAC design. The design flow is stored in the form of a graph data structure.

IV. Prototype Implementation on WARP Board

We have implemented our framework on WARP v1 boards [2] using the OFDM Reference Design v.14. WARP board v1 is a popular SDR development platform with Virtex Pro II FPGA and two PowerPC cores synthesized on the FPGA. Our fundamental MAC components are implemented on the FPGA or software on PowerPC depending on their timeliness requirements, computational needs and the degree of usage. We use virtualization and individual resource management of these components, which allow efficiently implementing complex MAC designs on the platform. The rest of our toolchain is implemented in the software running on the PowerPC. It should be noted that our tool itself is not WARP specific, but can be easily ported to other SDR platforms.

V. Experimental Evaluation

In this section we describe the code reuse across different MAC implementations on our framework and the reconfiguration response time and memory footprint of our prototype implementation.

TABLE II
Component reuse for MAC realizations on WARP [1].

Component	Aloha	Pure_CSMA	B-MAC	IEEE 802.11	S-MAC	CogMAC
Send Packet	0	0	1	3	4	5
Expect Frame	0	0	1	2	3	4
RTS/CTS/DATA/ACK	0	0	0	1	1	0
Random No. Gen.	0	0	0	0	1	2
Timer	0	0	2	3	2	5
Carrier Sensing	0	1	1	1	1	4
Channel Switching	0	0	0	0	0	7
Radio States	0	0	2	2	3	4
SendFrame	1	1	0	0	0	1
ReceiveFrame	1	1	0	0	0	0

A. Component Reuse

Table II lists the MAC component reuse in realizing Aloha, simple CSMA, B-MAC [28], IEEE 802.11, S-MAC [29] and CogMAC [30] using our framework on the WARP boards. The gray shaded components represent the primary components while the unshaded components are the secondary level components. It can be observed that certain components are used multiple times within a particular protocol, which advocates the idea of realizing the key atomic functionalities in the hardware for increasing computing and communication efficiency. From the perspective of a user, a MAC becomes simpler to implement if the framework provides higher support for secondary level components. We have realized CogMAC [30], a decentralized spectrum agile cognitive MAC protocol with fairly complex multichannel operation in a fast and easy way using the MAC-meta compiler approach.

B. Execution Time Overhead

We have measured the protocol execution time using our toolchain in comparison to monolithic implementation on WARP board. User interaction with WARP board is carried out over the serial communication interface. However, our measurements do not take serial communication delay into account.

TABLE III shows the execution time of three MAC protocol implementations. Simple Aloha uses `Radio_To_Rx()`, `WriteToTxBuffer()` and `TxPacket()` while additionally CSMA MAC protocol includes `CarrierSensing()` and Spectrum-Agile MAC (CogMAC) protocol [30] uses `setFrequencyChannel()` and `ReadRssi()`. The packet transmission measured in this experiment is with QPSK modulation scheme for both

TABLE III
Execution time of different protocol implementations based on a component based PHY/MAC framework on WARP board with and without our toolchain [24].

Protocol	*Aloha*	*CSMA*	*CogMAC*
Number of components in the list	5	11	15
Execution time w/o toolchain [ms]	1.491	1.503	1.537
Execution time with toolchain[ms]	1.495	1.518	1.548
Execution time overhead (%)	0.27	1.0	0.72

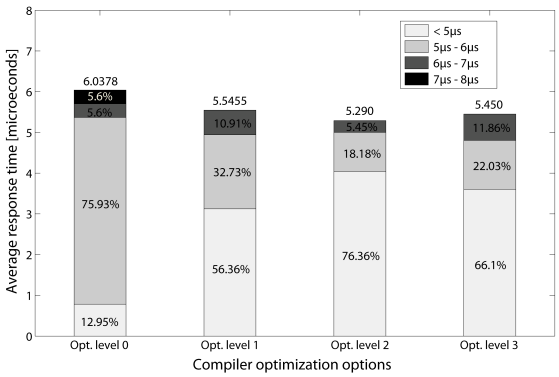

Fig. 6. Response time for MAC auto-configuration on WARP boards [1].

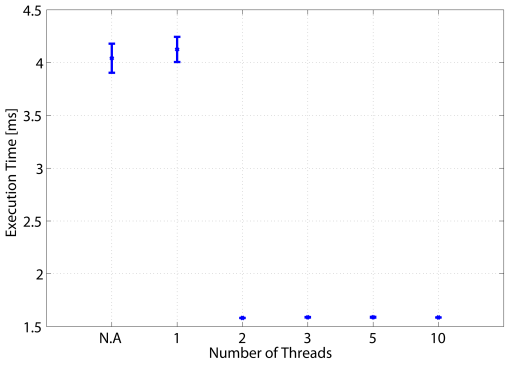

Fig. 7. Execution time with and without our toolchain on a single-core Linux machine with different number of threads [24].

the header and the payload. The lengths are 24 bytes and 1000 bytes respectively. We show that with three different implementations, the overhead caused by using our toolchain for servicing the node, locating the function, etc., is within the 1 % bound in terms of execution time.

C. Response Time Overhead

A fast response time is achieved through the function pointer based implementation approach followed in the Wiring Engine. Fig. 6 shows the average time required for a MAC to reconfigure itself. The graph shows the average results for 10 000 readings at different optimization levels. Here we consider the elapsed time for the Wiring Engine to update the state-machine of the MAC upon receiving a new configuration. The bar graph also shows the percentages of the individual reconfiguration delays. Please note that since the execution time for different components is different, here we consider the case when a configuration is updated (one component removed or inserted) instantly without waiting for the completion of the currently executing component. In other words, currently executing component has an "independent" entry in its dependency table [1] [24]. Our results show a fast reconfiguration response, which makes it suitable for designs with time-critical requirements. In realistic MAC implementations and auto-reconfigurations, more than one elements have to be re-wired [24]. We have observed a typical protocol reconfiguration time to be $\leq$ 1 ms.

D. Parallelization and Multi-core Efficiency

Multi-core architectures are nowadays common in general purpose and 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 network protocol execution, we explore the possible performance improvement we can benefit from multi-core platforms [23]. Unfortunately none of our currently available hardware platforms has multi-core architecture nor supports multi-threading. Therefore, in order to investigate the implications of multi-threading and parallelization in terms of execution speed, we have simulated test cases using up to 16 cores on a Linux based PC with multi-processors. The execution time of each MAC component is measured on WARP board and is used in the experiment to give a more educated guess on how parallelization can benefit network protocol execution when the hardware platform provides parallelization capability. A four-function protocol is used for our benchmarking purposes:

```
Radio_to_Tx();
WriteToTxBuffer();
ReadFromRxBuffer();
TxPacket();
```

The parallel dependency is defined as:

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

Fig. 7 shows the execution time for the program on a single-core machine when different numbers of threads are used. Since on PC the CPU is shared with other OS processes, single thread execution time is much worse than execution time on

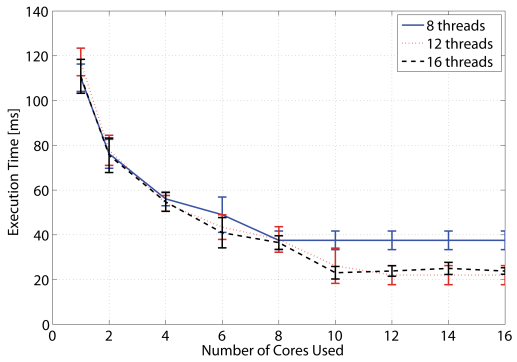

Fig. 8. Execution time of 16 independent tasks for computing mean and standard deviation over 10 000 samples per task using different number of threads on different number of processing cores [24].

WARP. When more resources are allocated with larger number of threads, the performance improved over 600 %. Fig. 8 shows a clear relationship between the number of processing units and the execution time. The optimum performance is achieved when 16 tasks are scheduled onto 12 cores instead of 16 due to the overhead for thread scheduling. Since the performance does not scale linearly with resource availability, there is a potential in optimizing the resources assigned depending on the nature of the tasks.

VI. Conclusions

In this paper, we have described a compiler assisted MAC design approach for component oriented frameworks. A domain specific MAC language is designed to express the entire MAC protocol with a few lines of code. The MAC description language models fine-grained MAC processes and enables portable designs. A host compiler translates the MAC code to be executed by the Wiring Engine. Composing a MAC protocol based on MAC components through our toolchain allows run-time reconfiguration which is highly desirable for dynamic environments, for instance, in the context of cognitive radios paradigm. Our prototype implementation on WARP SDR boards indicates that flexible and versatile MAC designs are achievable without significant loss of performance characteristics. We have observed that the additional execution speed overhead of the compiler assisted implementation for Aloha, simple CSMA and a spectrum agile cognitive MAC protocol [30] remains within 1 % bound compared to the implementations without using the toolchain.

Acknowledgment

We would like to thank the financial support from RWTH Aachen University and DFG (Deutsche Forschungsgemeinschaft) through the UMIC research center and EU through 2PARMA project grant. We would also like to thank Guangwei Yang and Marina Petrova for useful discussions.

References

[1] J. Ansari, X. Zhang, A. Achtzehn, M. Petrova and P. Mähönen, "A Flexible MAC Development Framework for Cognitive Radio Systems," in *Proceedings of the IEEE Wireless Communications and Networking Conference*, 2011.

[2] A. Khattab, J. Camp, C. Hunter, P. Murphy, A. Sabharwal and E. W. Knightly, "WARP: a flexible platform for clean-slate wireless medium access protocol design," *SIGMOBILE Mobile Computing Communication Review*, vol. 12, no. 1, pp. 56–58, 2008.

[3] J. R. Polastre, "A unifying link abstraction for wireless sensor networks," Ph.D. dissertation, Berkeley, CA, USA, 2005.

[4] K. Rerkrai, J. Riihijärvi, M. Wellens and P. Mähönen, "Unified Link-Layer API Enabling Portable Protocols and Applications for Wireless Sensor Networks," in *Proceedings of the IEEE International Conference on Communications*, 2007.

[5] G. Bianchi and A. Campbell, "A programmable MAC framework for utility-based adaptive quality of service support," *IEEE Journal on Selected Areas in Comm.*, vol. 18, no. 2, pp. 244–255, Feb. 2000.

[6] C. Doerr, M. Neufeld, J. Fifield, T. Weingart, D. Sicker, and D. Grunwald, "MultiMAC - an adaptive MAC framework for dynamic radio networking," in *Proc. of the IEEE DySPAN*, 2005.

[7] D. G. Messerschmitt, "Rethinking Components: From Hardware and Software to Systems," *Proceedings of the IEEE*, vol. 95, no. 7, pp. 1473–1496, 2007.

[8] R. Braden, T. Faber, and M. Handley, "From protocol stack to protocol heap: role-based architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 1, pp. 17–22, 2003.

[9] A. Sharma and E. M. Belding, "FreeMAC: Framework for multi-channel MAC development on 802.11 hardware," in *Proc. of the ACM workshop on Programmable routers for extensible services of tomorrow*, 2008.

[10] M.-H. Lu, P. Steenkiste, and T. Chen, "FlexMAC: a wireless protocol development and evaluation platform based on commodity hardware," in *Proceedings of the third ACM International Workshop on Wireless Network Testbeds, Experimental Evaluation and Characterization*, 2008.

[11] A. Sharma, M. Tiwari and H. Zheng, "MadMAC: Building a Reconfigurable Radio Testbed Using Commodity 802.11 Hardware," in *Proceedings of the IEEE Workshop on Networking Technologies for Software Defined Radio Networks*, 2006.

[12] P. Djukic and P. Mohapatra, "Soft-TDMAC: A Software TDMA-Based MAC over Commodity 802.11 Hardware," in *Procedings of the IEEE International Conference on Computer Communications*, 2009.

[13] B. Han, A. Schulman, F. Gringoli, N. Spring, B. Bhattacharjee, L. Nava, L. Ji, S. Lee, and R. Miller, "Maranello: practical partial packet recovery for 802.11," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, 2010.

[14] E. Blossom, "GNU radio: tools for exploring the radio frequency spectrum," *Linux Journal*, no. 122, p. 4, 2004.

[15] "The USRP Board," https://radioware.nd.edu/documentation/hardware/the-usrp-board [Last visited: 1st Nov, 2010].

[16] M. Neufeld, J. Fifield, C. Doerr, and A. Sheth, "SoftMAC: A Flexible Wireless Research Platform," in *Proceedings of the 4th ACM workshop on Hot Topics in Networking*, 2005.

[17] G. Nychis, S. Seshan and P. Steenkiste, "Enabling MAC Protocol Implementations on Software-Defined Radios," in *Proceedings of the Symposium on Networked Systems Design and Implementation*, 2009.

[18] K. Tan, H. Liu, J. Zhang, Y. Zhang, J. Fang, and G. M. Voelker, "SORA: high-performance software radio using general-purpose multi-core processors," *Communications of the ACM*, vol. 54, pp. 99–107, January 2011.

[19] C. Hunter, P. Murphy and A. Sabharwal, "Real-time testbed implementation of a distributed cooperative MAC and PHY," in *Proceedings of the IEEE Conference on Information Sciences and Systems*, 2010.

[20] P. K. Mckinley, S. M. Sadjadi, E. P. Kasten and B. H. C. Cheng, "A Taxonomy of Compositional Adaption," Michigan State University, MSU-CSE-04-17, Tech. Rep., 2004.

[21] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Transactions Computer Systems*, vol. 18, no. 3, pp. 263–297, 2000.

[22] V. Ramakrishnan, E. Witte, T. Kempf, D. Kammler, G. Ascheid, R. Leupers, H. Meyr, M. Adrat, and M. Antweiler, "Efficient and portable SDR waveform development: The Nucleus concept," in *Proceedings of the Military Communications Converence*, 2009.

[23] C. Silvano *et al.*, "2PARMA: Parallel Paradigms and Run-Time Management Techniques for Many-Core Architectures," in *Proceedings of IEEE Computer Society Annual Symposium on VLSI*, 2010.

[24] X. Zhang, J. Ansari, G. Yang and P. Mähönen, "TRUMP: Supporting Efficient Realization of Protocols for Cognitive Radio Networks," in *Proc. of the IEEE Symp. on New Frontiers in Dynamic Spectrum*, 2011.

[25] H. S. Lichte, S. Valentin, and H. Karl, "Automated development of cooperative MAC protocols: A compiler-assisted approach," *Mobile Networks and Applications*, vol. 15, no. 6, pp. 769–785, 2010.

[26] "The Lex & Yacc Page," http://dinosaur.compilertools.net/ [Last visited: 1st Nov, 2010].

[27] E. M. Nahum, D. J. Yates, J. F. Kurose, and D. Towsley, "Performance issues in parallelized network protocols," in *Proc. of the 1st Symp. on Operating Systems Design and Implementation, Usenix*, 1994.

[28] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *Proceedings of the International Conference on Embedded Networked Sensor Systems*, 2004.

[29] W. Ye, J. Heidemann and D. Estrin, "An energy-efficient MAC protocol for wireless sensor networks," in *Proceedings of the IEEE Conference on Computer Communications*, 2002.

[30] J. Ansari, X. Zhang and P. Mähönen, "A Decentralized MAC for Opportunistic Spectrum Access in Cognitive Wireless Networks," in *Proceedings of the ACM SIGMOBILE Workshop on Cognitive Wireless Networking*, 2010.