TRUMP: Supporting Efficient Realization of Protocols for Cognitive Radio Networks

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

***Abstract*— Cognitive radios require fast reconfiguration of the protocol stack for dynamic spectrum access and run-time performance optimization. In order to provide rapid on-the-fly adaptability of PHY/MAC protocols, we have designed and implemented TRUMP: a Toolchain for RUn-tiMe Protocol realization. It includes a meta-language compiler, logic controller and an optimizer. TRUMP allows run-time realization and optimization of cognitive network protocols for the requirements of a particular application, communication capabilities of the radio, the current spectrum regulation and policies. TRUMP supports efficient multi-threading for multi-core platforms in order to meet variable computational requirements and to allow parallelization of PHY/MAC processing for cognitive radio systems. We have carried out the performance evaluation for different metrics on WARP SDR platform and embedded Linux based PCs. Our results indicate that TRUMP allows reconfiguration of protocols in the order of a few microseconds through run-time linking of different components, thus meeting the strict timeliness requirements imposed by PHY/MAC processing.**

I. Introduction

Over the past years, wireless technology has gained much popularity due to its low cost, ease of use and wide range end user services. Wireless communication is becoming an integral part of our daily life as development of smart phones and other mobile devices continues. As a result, the demand for spectrum has been continuously rising. However, the spectrum is a limited resource and wasteful use would lead to depletion and thus hinder the growth of wireless services. Cognitive radios are regarded as the technology means to fully use the available wireless spectrum. Mitola [1] anticipated that cognitive radios would become aware of the context and environment, and thus would be able to optimize their behaviour. The opportunistic use of frequency bands via cognitive radio principles is an attractive solution to decrease spectrum shortage without compromising the service quality for licensed users.

The unique characteristics of cognitive radio networks have imposed new challenges on architecture and protocol design, especially for the PHY and MAC layers. We will discuss these challenges in the following paragraphs.

First, cognitive radio has the ability to sense, learn and be aware of the following [2]: parameters related to the radio channel, availability of spectrum and power, the radio's operating environment, user requirements and applications, available network infrastructures, local policies and operating restrictions. As a premise, PHY and MAC layers designed for cognitive radio should be able to adapt themselves according to the prior knowledge and current observations so as to offer the optimal performance suited to application requirements. Classically, PHY and MAC-layer protocols are designed to be highly optimized for a particular application's use in a static environment. This approach contradicts to the requirement of cognitive radio networks. Therefore, a new paradigm in designing adaptable and flexible networking protocols with the capability of modifying PHY/MAC is required to unleash innovations and increase network performance.

Second, cognitive radios usually employ sophisticated sensing mechanisms and signal processing algorithms which require high computational power. The current trend in processor architectures for wireless terminals is towards multi-core processors connected by an on-chip network. To be able to fully utilize a multi-core architecture would intuitively bring enormous benefits in addressing requirements in terms of computational capability and timeliness for cognitive radio terminals. We have an on-going work towards flexible multi-core architectures in the context of a *Nucleus* project [3]. In this context, we have proposed a library based approach which enables efficient utilization of multi-processor systems. Computationally demanding PHY/MAC components with time critical requirements that capture common functionalities within the networking protocol stack are identified as Nuclei. These Nuclei can be executed in parallel.

Third, compared to the traditional radio technologies and network protocols, cognitive radios require close interaction among network layers, especially between PHY and MAC layers, in order to meet the highly demanding timeliness requirements, the different application demands, the different capabilities of the mobile user terminals, etc. Learning and optimization at run-time are to be achieved not only through traditional cross-layer design by exchanging information and changing parameter passing between different network layers; the ability to change the composition of protocol stack and thus modify the actual protocol behaviour is also of high interest for cognitive radios and is addressed in this paper. TRUMP is an integral part of our general architectural work towards self-optimizing cognitive radio architecture and implementations. We have earlier introduced Cognitive Resource Manager (CRM) architecture [4], and this architecture calls for flexible, runtime-optimizable MAC layer. TRUMP is our attempt to provide such solution for CRM, but it is specifically designed to be architecture and processor independent approach so that it can be freely used even for independent PHY/MAC development work.

978-1-4577-0178-8/11/$26.00 ©2011 IEEE

The major contributions of our work include:

1) We have designed and implemented a lightweight toolchain, TRUMP, which allows fast protocol realization. TRUMP enables both user and self-triggered reconfiguration strategies for protocol compositions at run-time with very low latency. Our experimental results show that the configuration delay using our tool is in the order of a few microseconds on commercially available SDR platforms and low power embedded wireless nodes. This makes it possible for the protocol stack to meet the requirement for adaptability, flexibility and strict timelines of cognitive radio networks. In order to realize run-time composition with minimum user-effort, an efficient language syntax for PHY/MAC protocol description and the associated compiler are designed. The language has simple syntax but is expressive enough to realize and compose any complex protocols. It eases user implementation effort and offers the possibility of automating code generation and adaptation with cognition.
2) In order to allow parallelization and prioritization of different processes, TRUMP provides a scheduler for multi-threading operations. Our results show that having the ability to run efficiently on multi-core platforms can lead to over 80 % gains in execution speed.
3) The efficient reconfiguration capability of TRUMP is combined with a run-time performance optimizer. Our toolchain enables protocol optimization based on user-preferences, e.g. protocol memory size, execution speed, power consumption or a combination of the above. Our experimental results based on PHY/MAC performance evaluation on a COTS SDR development board indicate that we are capable of optimizing performance through both parameter settings and modifications of components for protocol compositions. The integrity of the program logic and data flow are guarded by our system that no erroneous protocol realizations will be executed.

Our toolchain is lightweight, does not require heavy compilation during runtime and can easily be deployed on a wide range of COTS platforms. We have extensively evaluated it on WARP [5] boards, and embedded Linux PCs. Various COTS embedded sensor nodes including TelosB, TelosA, MICAZ, MICA2, MICA2dot and IntelMote2 are used for stress testing our implementation with very low computational power and energy budget devices. We have proven that the TRUMP approach is easily portable and lightweight enough for embedded devices which can be used as distributed spectrum sensing platforms [6]. Our experimental results show that protocols realized through TRUMP obtain a 2 % saving on power consumption. In our test scenarios we are able to demonstrate nearly 400 % increase in throughput due to the flexibility to change PHY/MAC layers in run-time as described in Section V. Although TRUMP adds a memory overhead of approx. 10 % to the protocol stack, we think that this is a minor cost against all the benefits.

The rest of the paper is organized as following: In Section II, we present the related research work in this area and show the relevance and novelty of our work. The design details of TRUMP are shown in Section III and Section IV gives a brief description of the hardware platforms on which we have implemented and deployed TRUMP. Detailed evaluation of our work based on empirical studies is provided in Section V and finally we conclude and discuss about future research directions in Section VI.

II. Related Work

Classically, protocols are implemented in a monolithic fashion with tight coupling to the underlying hardware platform, which restricts the aspects of reconfigurability required in highly dynamic and agile systems such as cognitive radio networks. One common approach to introduce adaptivity is through appropriately setting the parameters of a particular protocol [7]. However, the scope of this approach is limited by the nature of that protocol and the permitted range of parameters. In order to expand the scope, Doerr *et al.* proposed the idea of switching among a few pre-defined standalone protocols as per the changing application requirements [8]. Based on this idea, AMAC [9] has been implemented using GNU radio platform which enables switching between CSMA and TDMA protocol based on the traffic variations in the network. However, we argue that though this scheme provides flexibility and adaptability features, the design choice is limited to a subset of pre-selected protocols and it can at best approximate the closest fit, let alone the fact that many communication systems and devices are limited in terms of the onboard memory.

Component oriented protocol designing [10] has been well investigated and many solutions have been proposed and implemented to allow flexibility as discussed by Braden *et al.* [11]. Bianchi *et al.* discuss the advantages of adaptive and programmable MAC schemes, which satisfy changing application demands [12]. The approach of implementing PHY/MAC functionalities in pure software provides the required flexibility and adaptability for cognitive radio applications, but fails to meet the strict time critical requirements [13]. USRP and GNU Radio [14] platforms provide means for flexible protocol implementations. However, being entirely software components running in the general purpose CPU, they are not able to meet the performance characteristics as has been reported by other researchers [13], [15]. The Sora platform [16], by exploiting sophisticated parallel programming techniques on multi-core CPU architectures, achieves throughput comparable to IEEE 802.11 hardware. Although the PHY processing blocks are implemented in software, the platform is augmented with a radio control board with FPGA for time-critical portions of the protocol.

Sharma *et al.* designed the FreeMAC [17] – a multichannel MAC development framework exposing the PHY/MAC interfaces for the protocol developer on top of the standard IEEE 802.11 hardware. This work aims at supporting channel switching and allows better timeliness characteristics for spectrum agile MACs. A similar work has been demonstrated

in [18]. Sharma *et al.* [19] control the frame timings at the transmit and receive interfaces through the MadWifi driver for Atheros based NICs in order to improve the overall throughput. However, since IEEE 802.11 hardware is restricted in providing granular accessibility to radio and PHY parameters as needed by spectrum agile and cognitive networks, the configurability aspects remain limited. Furthermore, these designs are not implemented in a modular fashion – only certain PHY/MAC parameters can be controlled instead of being able to reconfigure PHY/MAC protocol composition at a much fine grained level. Airblue [20] proposes a system where both PHY and MAC are implemented in FPGA in a modular fashion to achieve low latency cross-layer communication to facilitate cross-layer wireless protocol development. PHY layer processes can be configured at runtime by control tokens passing through the PHY module pipeline. Although achieving a fast request-response between PHY and MAC, the modules provided by the system are too simple to handle complex PHY/MAC realization required for cognitive radio networks. Dutta *et al.* on the other hand, have proposed an architecture specifically for software defined cognitive radio [21] which provides only the barebones of a OFDM based radio physical layer for easy adaptation.

In order to realize efficient MAC implementations, Nychis *et al.* [13] have split the MAC functionalities based on the timing characteristics and their latency tolerances. Following the same design methodology, we have proposed Decomposable MAC Framework [22] [23], which partitions the PHY/MAC component implementation on software and hardware in a way that time critical components reside in the hardware for computing and communication speed gains while flexible binding of these PHY/MAC components and decision logic can be achieved in the software. Unlike the approach of Nychis *et al.*, we have defined the PHY/MAC components based on the timeliness requirements, communication demands and the degree of component reuse. This way these components can serve as the building blocks for a wide range of PHY/MAC protocols for cognitive radio networks. In particular, we have demonstrated the implementation of a few spectrum agile MAC solutions based on the framework [22].

The concept of component interconnections has been long established in software engineering domain 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 [24] because of the need for run-time reconfigurations, such as in robotic software systems. Networking researchers have actively used Click [25] for binding components together however in order to allow fast speed and suit to limited resources, customized solutions are more suitable as we show later in our evaluation in Section V. In our earlier work [23], we have shown that running customizable tool consumes only little memory on the WARP SDR platform and give a response time in the order of a few microseconds.

We have followed the approach of binding PHY/MAC building blocks at run-time in the Nucleus [3] flagship project of the UMIC Research Center and EU funded 2PARMA project [26]. The PHY/MAC components requiring intensive communication and computation are identified to be realized on the dedicated cores on the multi-core platform or different ASIPs. Efficient communication interface architectures have been proposed to fully utilize the parallelization of processes through appropriate resource management schemes and state-machine execution controllers. We believe that our approach opens a wider experimental room and perspective for dynamic protocol realization for cognitive radio networks. In order to benefit from this approach, a supporting toolchain is required to exploit the platform architecture for achieving highly efficient and dynamic solutions. TRUMP allows realization of a wide range of protocols at run-time from a set of basic components used as the building blocks and enables dynamic adaptation of PHY/MAC behaviours to the changing spectral conditions and application requirements. Unlike the approach [8] of switching among different protocols and requiring to store the perceivable solutions beforehand in the memory, TRUMP consumes relatively only small memory as it allows the composition of different protocols from the same set of components when needed, for instance, with the evolving application requirements and policy constraints, etc. A compiler assisted approach of developing MAC protocols as discussed in [27] provides a systematic way of automation of MAC implementation, analysis and code generation. However, it does not address the timeliness demands in real-time cognitive PHY/MAC stacks.

III. System Design

In this section, we describe the design details of our toolchain for realizing protocols for cognitive radio networks. Our toolchain consists of five parts: meta-language descriptor, compiler, logic controller, optimizer and a wiring engine. Fig. 1 illustrates the comparison between the classical methodology and TRUMP approach for protocol development and realization cycle. We shift the implementation of protocol on an SDR development board from the user end to board side and provide a means for run-time command/response based communication between the end user and the running protocol in order to achieve simplicity, adaptability and flexibility. The design and implementation of this toolchain is carried out using bottom-up approach as orderly described in this section and it pre-requires that a component-based framework of the target protocol group such as [23] for MAC protocol realization is available.

A. Component-based Framework

Although defined protocol stacks work well in fixed networks, performance in terms of throughput, latency, etc., of wireless networks is severely affected by the stringent layering since it is difficult to optimize the network stack for dynamic application scenarios. Designing protocols in a modular way is therefore very promising [8] [10] [12]. Our designed system works on component-based framework where both input and output of the components are clearly defined. As shown in [22]

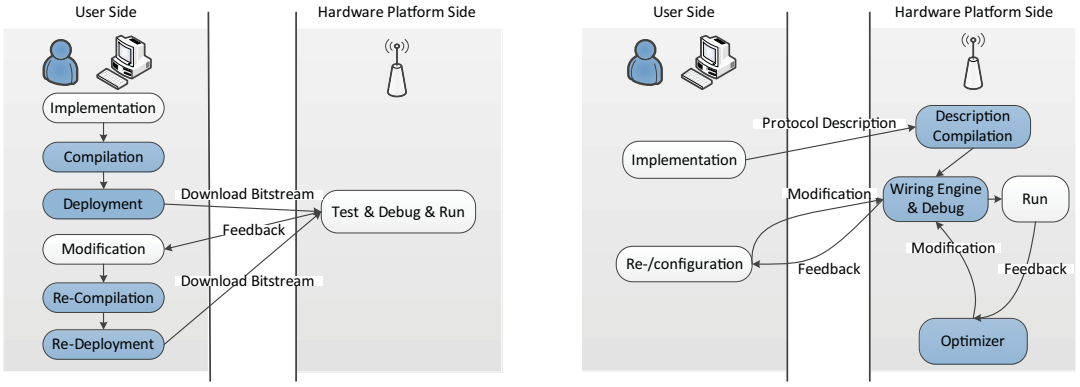

Fig. 1. Comparison between existing protocol development process and the solution proposed and implemented in this paper.

TABLE I
DEFINITIONS FOR INSTRUCTIONS USED IN TRUMP.

Instruction	*Parameter*	*Functionality*
VAR	Variable ID	Allocates new memory for a variable
FUNC	Function Pointer	Calls a pointed Function
SET	Variable ID, Constant or Function Pointer	Sets either a constant value or the returned value of a Function to a variable
IF	Function Pointer	If the Function returns FALSE, jump to the next node of the matched ELSE
ELSE	NULL	Jumps to the matched END
END	NULL	NULL
GOTO	Label ID	Jumps to the matched LABEL
LABEL	NULL	NULL

and [23], MAC protocols can be decomposed into elementary components such as *Carrier Sensing*, *Timer*, *Radio Switching*, etc. Similarly, PHY layer components such as *FFT*, *IFFT*, *Modulator*, *Demodulator*, *Channel Estimator*, etc can be used as building blocks for PHY processes.

B. Wiring Engine

In order to correctly bind components together and execute the composed protocol 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.

Wiring engine defines a unified API for all the functions and logical connections. A linked list structure is used to store the composition of protocols in terms of both functional and logical components defined in an instruction set described later. A run-time execution manager takes care of the execution flow of the protocol. Details of each component of the engine are described in the following paragraphs.

1) Component Interfaces: In our implementation, all the interfaces have a standard format which takes an integer pointer as parameter and returns an integer, e.g. `int SendPacket(int* para)`.

2) Linked List: We employ linked lists to realize desired protocols by re-/linking the components together to meet the requirement for dynamic adaptation. Since all the components share a unified API, we can easily use a function pointer to address a 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 dynamicly adding/deleting/reordering of components while being easily manageable. Thus, run-time composition of protocols is realized. A conventional array is also an option for sequential data storage; however, it is more difficult to maneuver for dynamic adaptation of array connections.

3) Instruction Set: We define a set of instructions to realize logical and non-sequential operations which are necessary in realization of protocols. In order to minimize implementation complexity and memory overhead as required by embedded platforms in general, we design a small instruction set as shown in TABLE I, which is sufficient for complex protocol realization. The instructions are represented as the elements in the linked list and running an application in the form of linked list is done by traversing the whole list from head to tail.

4) Run-time Execution Manager: Since logical operations are included in the nodes, the sequence of program execution involves run-time feedback and decision making. Therefore, we design a run-time execution manager which decides on the next node to be executed on-the-fly using decision logic described in Algorithm 1. It is simply a switch statement which depending on the keywords defined in the instruction set, sets the pointer to the next element to be executed accordingly.

TABLE II
TYPES AND DEFINITION OF DEPENDENCIES IN TRUMP.

Dependency Type	*Representation*	*Definition*
Hard dependency	$f_a(\prec\|\succ)f_b$	f_b must be present for f_a to be executed in defined order.
Weak dependency	$f_a(< \mid >)f_b$	f_a can be executed independently, but if f_b is present then two must be executed in defined order.
Parallel dependency	$f_a \parallel f_b$	f_a and f_b can be executed concurrently.

```
Data: linked list
InstructionSet = VAR, FUNC, SET, IF, ELSE, END, GOTO,
LABEL;
CurrentNode = List.head;
while CurrentNode ≠ List.tail do
    switch CurrentNode.type do
        case VAR
            stack.add(CurrentNode.resource);
        case FUNC
            call CurrentNode.resource;
        case SET
            set CurrentNode.resource;
        case IF
            if CurrentNode.resource = TRUE then
                do nothing;
            else
                CurrentNode = match(ELSE).next;
            end
        case ELSE
            CurrentNode = match(END).next;
        case END
            do nothing;
        case GOTO
            CurrentNode = match(CurrentNode.resource);
        case LABEL
            do nothing;
        otherwise
            report ERROR;
        endsw
    endsw
    CurrentNode = currentNode.next;
end
```

Algorithm 1: Run-time execution manager makes decisions based on the contents of the list nodes.

C. Logic Controller

The logic controller acts as the brain of the toolchain. It ensures smooth execution of the composed protocol by identifying conflicts between the relationship among components and the execution order.

1) Dependency Management: The wiring engine 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, we have devised pre-defined rules to express interdependencies among components and incorporate them as part of the functional libraries. There are two rules for the design decisions:

1) **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 block.
2) **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 in total three types of dependencies which can be expressed through our system as shown in TABLE II: A *hard dependency* is normally found in hardware-related system design. For example in a MAC protocol design, when a packet is to be sent, i.e. f_a is `radio_send()`, f_b would be `radio_on()` since without the radio being turned on, transmission of packets cannot be carried out. In object oriented programming, one can for example release an object only if it has been constructed already. This kind of dependency is usually pre-fixed and always true for the same type of target platforms. A *weak dependency* can usually be a user-defined common logical action. For example in MAC protocol designs, function `SendPacket()` can be executed as it is but if `CarrierSensing()` is present, then `SendPacket()` is executed after `CarrierSensing()`. Furthermore, if functions are independent of each other, they can be run concurrently. TRUMP allows designers at run-time to explicitly indicate *parallel dependency* between functions for parallel execution. For example `PrepPktToTx()` can be executed at the same time as `ReceivingPkt()`. User-defined dependencies cannot overwrite *hard dependencies*.

In order to store the defined dependencies, we have created a resource table which contains both function definitions and dependency information as shown in Fig. 2. Dependencies are given among functions and represented in the form of three-bit numbers, i.e. 000 means independent, 001 represents $f_a \prec f_b$ and 010 refers to $f_a \succ f_b$, 011 and 100 maps to $f_a < f_b$ and $f_a > f_b$ respectively while 101 indicates parallel dependencies. One resource table entry stores one API function definition and a bitmap which indicates all the dependency relationships with others in the table. Since each dependency is represented with 3 bits, $3*N$ bits are required in

Resource Table			
Index	Name	Pointer	Dependency Bitmap
1	Radio_Init	0x0A2B	000 | 000 | 000
2	SendFrame	0x0F42	101 | 000 | 001
3	ExpectFrame	0x0CC5	000 | 101 | 001

Fig. 2. An example of dependency storage in resource table.

total to represent all the dependencies for *N* entries in the table. The i^{th} group of 3 bits in the bitmap indicates a dependency relationship with the i^{th} function in the table. The lookup algorithm can be represented as:

$$\texttt{table[m].bitmap} \gg \texttt{(3 * (n - 1)))} \% \ 8,$$

which returns a 3-bit value representing a dependency relationship between the functions defined in the m^{th} and n^{th} entries. In Fig. 2, both `sendFrame()` and `ExpectFrame()` have *hard dependencies* on `Radio_Init()` while they are able to be run in parallel.

2) Parallelization: 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 brings. In order to realize and manage execution, we use a stack $Rstack$ to store the currently running function(s). Each time before a function is pushed to $Rstack$, the dependency between itself and all the functions in $Rstack$ are checked. If dependency restriction is found, the function can only be pushed to $Rstack$ when the conflicting function is popped out of the stack.

foreach $rf_i \in Rstack$ **do**
  if $rf_i \nparallel f$ **then**
    Wait until rf_i complete;
  end
end
Rstack.push(f);
call f;

Parallelization is realized by both hardware and software. If there are more than one processing unit (e.g. GPU, MPSoC, etc.) existing, functions can be scheduled onto different hardware units and executed simultaneously. In software realizations, multi-threading is used for this purpose. Benefit brought by parallelization is evaluated in Section V-C.

D. Optimizer

TRUMP offers the capability of run-time configuration and linking of components from functional libraries. An optimizer is designed to fully utilize the configuration capability for run-time performance optimization purposes.

Cognitive radio networks leverage heavily from the machine learning algorithms for run-time optimization and adaptation of protocol behaviour according to dynamic spectral environment. Our system is designed to support different kinds of learning algorithms such as Genetic Algorithm, Bayesian reasoning, etc. 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 currently executing linked list. We have implemented an optimizer with decision making capability based on different optimization goals. Parameters including packet delivery ratio, throughput, current consumption, modulation scheme and interference level are used in the optimizer. The details of the construction and performance gain from the enabled self-optimization in our experimental measurements are shown in Section V-B.2.

Prioritization is important when there is a limitation of resource, e.g. only one function can be executed at that instance when there is a set of independent paralleled functions to be executed. For example, although logically `PrepPktToTx()` can be executed at the same time as `ProcessRcvdPkt()`, only one processing unit is available. If the protocol aims at optimizing packet delivery latency, `ProcessRcvdPkt()` should be prioritized while if throughput is of a higher concern, `PrepPktToTx()` is to be executed first. Therefore, a user can define the exact priority among components in the form of $f1 \ll f2 \ll f3 \ll etc.$ In addition, the optimizer can decide on the schedule of execution order according to the current optimization goal. Furthermore, there are rule of thumb to be followed when there is no user specifications. For example, if we are aiming as optimizing towards overall execution time and the time required for three functions $f1$, $f2$, $f3$ are $t1$, $t2$ and $t3$ respectively where $t1{>}t2{>}t3$. In general, $f1$ is to be executed first in TRUMP since if there is more resource available during the course of or after running $f1$, the overall execution time is more likely to be shorter than scheduling the shortest task first. However, such type of batch processing problems requires overall more prior knowledge regarding resource availability in order to optimize scheduling. Therefore, prioritization is also treated as a learning process in our optimizer and the rules of scheduling are updated based on the run-time experience and feedback.

E. Meta-language Descriptor

In order to facilitate efficient protocol implementation, we have included the concept of meta-language and its supporting compiler which is to be introduced in Section III-F. We use a C-like language with keywords *if*, *else*, *endif*, *label*, *goto* corresponding to the instruction sets.

F. 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* [28]. 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.

1) 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 network protocol realizations, our interest mainly lies in network performances and hardware constraints. Therefore, we have designed optimization options shown in TABLE III. Compiler requires knowledge about execution time of each

TABLE III
COMPILER OPTIMIZATION OPTIONS.

Option	*Optimization Goal*
$-e$	Energy consumption
$-t$	Execution time
$-m$	Memory usage

individual functions, energy consumption at different protocol and hardware states, memory requirement of each single block and availability overall. If there are multiple implementations for the same functionality available and the user does not specify a choice, the compiler picks the implementation which is optimized for user preferences. For example, when a user sets $-e$ when compiling protocol without specifying the modulation scheme, TRUMP chooses QAM16 according to TABLE VIII in Section V. The choice is made at compile time and can be modified through the optimizer during the run-time when user has specified different preferences.

IV. IMPLEMENTATION PLATFORMS

WARP board v1 is a popular SDR development platform with Virtex Pro II FPGA. It offers a wide range of capabilities in using and modifying the radio/PHY functionalities. It has 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's MAX2829 RF transceiver. We have extracted a set of PHY/MAC functionalities to be used by TRUMP.

Low-power embedded networks are becoming more and more common in daily life applications. Many of these networks require run-time adaptation for spectrally efficient operation and to support different application demands. In the recent years, the research community has come up with innovative cognitive algorithms for mitigating wireless interferences in order to achieve higher spectrum utilization [29], [30]. We have implemented the wiring engine on commercially available resource constrained nodes such as TelosB, MICAz, Intelmote2 and MICA2 actively used in embedded wireless sensor networking. The radio stack implementation provides a rich set of functional library for implementing different intelligent MACs and spectrum agile solutions. Embedded sensor nodes in general have limited resource in terms of hardware capability. The Telos series platforms use MSP430 microcontroller, have 10 kB RAM and 48 kB flash memory. MICA2 uses ATMega128L microcontroller and has 4 kB RAM. Being GCC compliant, our toolchain has easily been adapted to run on resource constrained embedded wireless nodes.

V. EXPERIMENTAL PERFORMANCE EVALUATION

In order to evaluate the performance of TRUMP, we focus mainly on two aspects: system overhead in terms of memory usage and execution time. Our comparison base-lines are protocols designed and implemented in monolithic fashion. The evaluation is carried out on both WARP boards and sensor nodes. We have also evaluated the performance of our run-time optimizer on WARP boards as the boards offer sufficient support for capability for PHY/MAC parameter extraction and components modification. However, since WARP board does not support multi-threading and parallel execution of tasks, we have also used Linux based multi-core embedded PC to evaluate the multi-threading gain and the subjected overhead.

A. Memory Usage and Execution Time Overhead

We have carried out experiments to evaluate the realization of PHY/MAC algorithms based on the radio stack with and without TRUMP on different hardware platforms. In particular, we analyze the execution speed and memory footprint across different resource constrained embedded platforms.

1) WARP Boards: We have measured the memory footprint and protocol execution time using TRUMP in comparison to monolithic implementation without TRUMP on WARP board. User interaction with WARP board is carried out through serial communication. However, our measurement does not take serial communication delay into account. The executable ELF file for the PowerPC on the WARP board shows that TRUMP imparts approx. 10 kB code size which is small enough to be deployed on most of the COTS embedded platforms. TABLE IV shows the execution time of three MAC protocol implementations. Simple Aloha uses `Radio_To_Rx()`, `WriteToTxBuffer()` and `TxPacket()` while CSMA MAC protocol includes `CarrierSensing()` and Spectrum-Agile MAC (CogMAC) protocol [31] uses `setFrequencyChannel()` and `ReadRssi()`. The packet transmission measured in this experiment is with QPSK modulation scheme for both 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 TRUMP for servicing the node, locating the function, etc., is within the 1 % bound in terms of execution time.

In addition, the re-configuration delay and different command execution time are presented in Fig. 3 for different protocol complexities. *Add* and *Delete* statements add and delete nodes in the list and are commonly used commands in modifying protocol realizations. The measurements are taken when the 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, to show the worst timing performance scenario. The *Check* instruction performs logic check of the function list to report if there is any dependency conflict or missing

TABLE IV
EXECUTION TIME OF DIFFERENT PROTOCOL IMPLEMENTATIONS BASED ON A COMPONENT BASED PHY/MAC FRAMEWORK ON WARP BOARD WITH AND WITHOUT TRUMP.

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

TABLE V

MEMORY FOOTPRINT [BYTE] FOR VARIOUS EMBEDDED SENSOR NODES WITH AND WITHOUT TRUMP FOR AN APPLICATION GENERATING A CONSTANT BIT RATE TRAFFIC USING SIMPLE CSMA/CA BASED MAC PROTOCOL.

Platform	MICAz		MICA2		TelosB		TelosA		Intelmote2		Mica2dot	
	RAM	ROM	RAM	ROM	RAM	ROM	RAM	ROM	RAM	ROM	RAM	ROM
With TRUMP	1732	12578	1651	10540	1770	11984	1770	11830	2776	21804	1651	10404
Without TRUMP	294	11662	212	9422	314	10982	314	10828	388	20922	212	9320

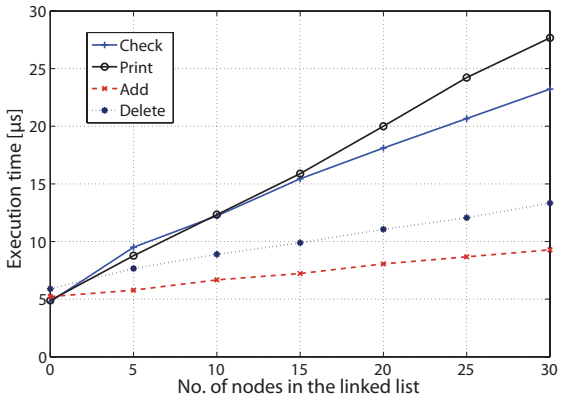

Fig. 3. Execution time when performing commands on the running list.

logic operators and *Print* is a debug friendly command which prints the content of the current list. Since all the command requires servicing the entire linked list, the execution time is hugely dependent on the size of the list. We see that the time increases linearly with the number of nodes in the scale of microseconds. Our run-time optimizer uses the same set of commands to change protocol composition and the execution time is identical.

2) Sensor Node Platforms: TABLE V shows the memory footprint in terms of RAM and ROM consumption for the whole radio stack with and without TRUMP for various embedded sensor node platforms. The table shows that the framework has only less than 10 % increased overhead in terms of ROM usage. It is worth noting that for these devices, the approach of run-time loading of different static PHY/MAC protocols [8] would be limited to a small set of selection choices due to the sheer memory constraints. In contrast, with our approach which allows composition of PHY/MAC protocols at the run-time, much wider design choices are available for fulfill the desired application requirements. TABLE VI shows the execution speed of variable assignment as an example on different embedded node platforms. The table also indicates the initialization overhead which is paid only once.

TABLE VI

EXECUTION SPEED [μS] OF ONE VARIABLE ASSIGNMENT FUNCTION WITH AND WITHOUT TRUMP ON VARIOUS EMBEDDED NODE PLATFORMS.

Platform	*MICAz*	*MICA2*	*TelosB*	*TelosA*
Static function execution	0.550	0.549	1.327	1.311
Function execution with TRUMP	15.221	14.995	32.995	32.000
Function loading	547	534	1241	1232
Function list initialization	838	831	2072	2001

B. Run-time Optimization

In this section, we describe in detail the formulation of our run-time optimizer and the performance gain achieved.

1) PHY/MAC Parameter Values: As mentioned in Section III-D, prior comprehensive knowledge of all the components in the function library is not only beneficial but essential to our optimizer. We have measured the execution time a set of fundamental MAC and PHY processes on WARP as shown in Table VII to be fed in to the optimizer. The measurement is done using a Agilent Infiniium DSO8104A oscilloscope with sampling rate at 10 M/s. Our measurement setup is shown in Fig. 4.

The current consumption of WARP board v1 with one daughter board attached at different operating stages is measured using an Agilent N2783A current probe along the main power supply cable. The readings are listed in Table VIII. The base current in supporting WARP board without any PHY/-MAC radio activity is 526.5 mA. Fig. 5 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 VIII, 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

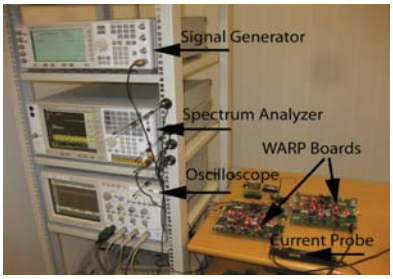

Fig. 4. Experimental Setup.

TABLE VIII

CURRENT CONSUMPTION FOR DIFFERENT OPERATIONS ON WARP BOARD.

WARP PHY/MAC Status	*Radio Status*	*Peripheral Servicing*	*Current consumption*
Initiliazed	Off	None	821.3 mA
Initiliazed	Off	Constant polling	838.5 mA
Initiliazed	Rx	Constant polling	861.8 mA
Initiliazed	Tx	Constant polling	932.1 mA
Initiliazed	Tx packet with zero payload at 100 packet/s	Constant polling	865.6 mA
Initiliazed	Tx packet with 1000 byte payload (BPSK) at 100 packet/s	Constant polling	891.7 mA
Initiliazed	Tx packet with 1000 byte payload (QPSK) at 100 packet/s	Constant polling	887.7 mA
Initiliazed	Tx packet with 1000 byte payload (16QAM) at 100 packet/s	Constant polling	879.8 mA

TABLE VII

PHY/MAC FUNCTION EXECUTION DURATION ON WARP.

Function	*Execution Time*
Radio_to_Sleep()	3 μs
Radio_to_Tx()	4 μs
Radio_to_Rx()	4 μs
Radio_Turnaround()	4 μs
SetFrequencyChannel()	22 μs
ReadRssi()	1.4 μs
CarrierSensing(1 sample,1 μs/sample)	3.7 μs
WriteToTxBuffer(1000 Bytes)	19.5 μs
ReadFromRxBuffer(1000 Bytes)	33.5 μs
CopyAmongRadioBuffers(1000 Bytes)	50.8 μs
CopyAmongPPCBuffers(1000 Bytes)	14.5 μs
TxPacket(1000 Bytes, BPSK)	1435 μs
TxPacket(1000 Bytes, QPSK)	758 μs

is almost double when BPSK modulation scheme is deployed as compared to QPSK. These facts are important to be used as prior knowledge for the optimizer for possible optimizations.

2) Modulation vs. Packet Delivery vs. Energy Consumption: Camp and Knightly have observed that for each modulation scheme, there is a range of SNR at which highest throughput is achieved [32]. 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 a Agilent E4438C 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 Fig. 6, 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. We incorporate our experimental readings as a priori knowledge for the optimizer as described in Section III-D. We set the packet delivery ratio requirement to be 60 % and are interested in best energy consumption performance. Fig. 7 shows the behaviour of a simple protocol

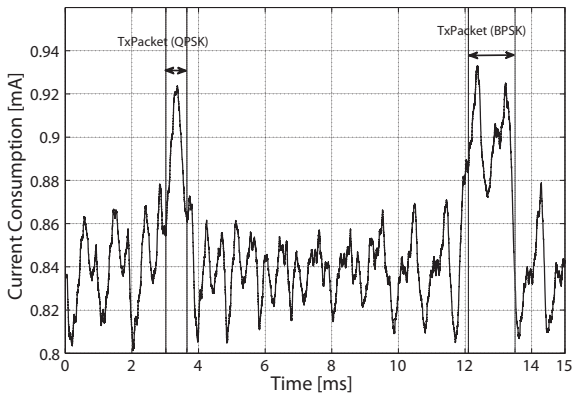

Fig. 5. Current consumption of WARP board. Packets with 24 byte header and 1000 byte payload with QPSK and BPSK modulation schemes are transmitted.

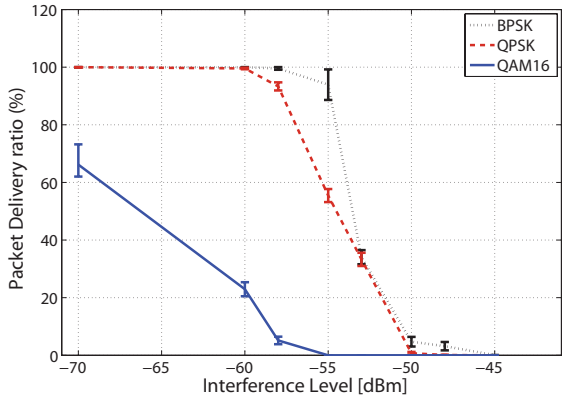

Fig. 6. Packet delivery ratio on WARP boards using different packet payload modulation schemes at different interference level. Packets are generated at 100 packet/s. The size of packet header and payload are 24 bytes and 1000 bytes respectively. Packet header is modulated with BPSK.

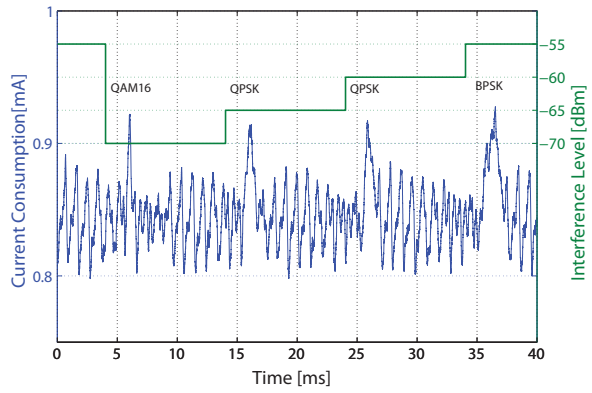

Fig. 7. Current consumption of WARP board with TRUMP energy consumption optimization under different interference levels.

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.

In addition to optimization through changing of parameters, we have enabled options of optimization through changing protocol structures by adding/deleting/re-wiring the components in the protocol description. Fig. 8 and Fig. 9 shows the performance of ALOHA and CSMA MAC protocols achieved on WARP board, measured between a single transmitter/receiver pair. CSMA MAC protocol can be formed based on ALOHA by simply adding a `CarrierSensing()` element, an `IF` and an `ENDIF` node. Our measurements show that with single channel single transmission flow, when the interference occupancy ratio is below 10 %, ALOHA is pretty efficient while when the interferer occupancy ratio is between 10 % and 35 %, a choice between throughput and packet delivery ratio is to be made. When the performance of either ALOHA or CSMA can suffice, we introduce a `ChannelSwitching()` component where a second channel which the transceiver supports is to be used. In our implementation, the overhead for switching channel and establishing link between the transmitter/receiver pair result in 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, either more efficient channel switching algorithm is formed or staying in one channel using simple ALOHA or CSMA protocols becomes more beneficial.

With this as prior knowledge, we have used a decision tree based optimizer and carried out our experiment by setting our optimization goal of maximizing throughput. The interferer occupancy radio in a particular channel is varied from 0 % to 80 % as shown in Fig. 10 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 `ReadRssi()` in our protocol realization since it is necessary to monitor the channel condition as an on-line input to the optimizer. The throughput was observed to be high when the channel is free and experienced an immediate drop when interferer is detected. For the first 15 seconds, the `CarrierSensing()` component is added and deleted from the protocol based on the confidence level of the channel condition and thus we see an unstable performance since the interferer appears only for short periods of time. When the interferer occupies the channel for a long enough period of time, the `ChannelSwitching()` element is included where an initial downturn of throughput is observed due to the switching overhead. However, since good channel condition is experienced afterwards, `ChannelSwitching()` and `CarrierSensing()` are gradually removed and the throughput performance is regained. We observe a performance gain up to 400 % at the end when the interferer occupancy ratio is up to 80 %.

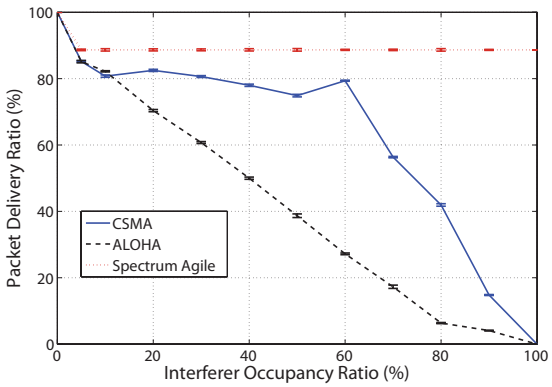

Fig. 8. Packet delivery ratio of ALOHA, CSMA and Spectrum Agile MAC protocols running on WARP.

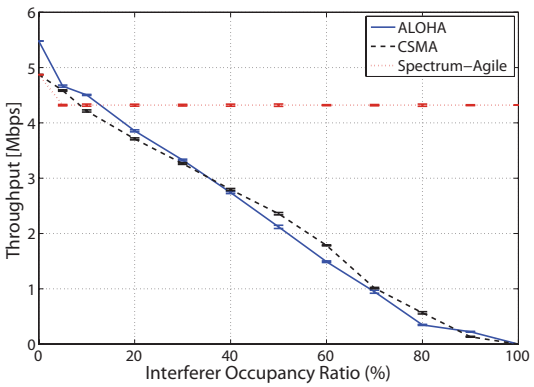

Fig. 9. Throughput of ALOHA, CSMA and Spectrum Agile MAC protocols running on WARP.

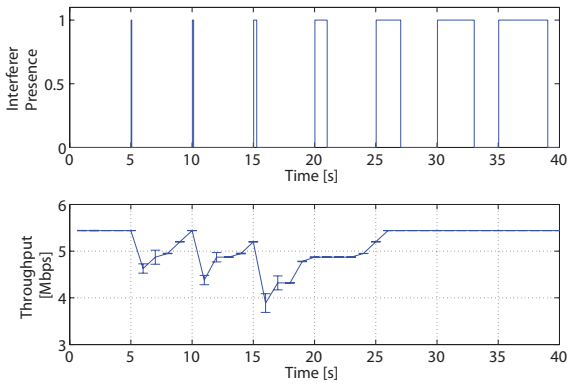

Fig. 10. The optimized throughput performance with the run-time optimizer.

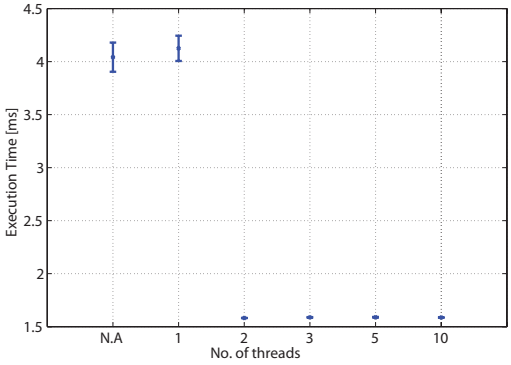

Fig. 11. Execution time with and without TRUMP on a single-core Linux machine with different number of threads.

C. Parallelization and Multi-core Efficiency

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 network 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 Linux based PC as our testbed. The readings measured on WARP board in TABLE VII are used to give a more educated guess on how parallelization can benefit network protocol executions when the hardware platform provides the capability. A four-function protocol is used for our benchmarkings purpose:

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

The parallel dependency is defined as:

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

Fig. 11 shows the execution time for the program on a single-core machine when different numbers of threads are used. Since on PC the CPU power is shared with other OS processes, single thread execution time is much worse than execution time on WARP. When more resources are allocated with larger number of threads, the performance improved over 600 %. Fig. 12 shows a clear relationship between the number of processing units and the execution time. 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 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 which is to be investigated in the future.

VI. Conclusion and Future Works

Cognitive radio network protocols require a high degree of flexibility and run-time adaptability in order to satisfy varying application demands and efficiently utilize the spectral resources. In this paper, we have described a toolchain, TRUMP, which allows run-time network protocol compositions by binding pre-defined protocol components together. Although run-time adaptation of component based systems has been well researched especially in computer science, our design specifically targets cognitive radios by providing fast reconfiguration speed and a lightweight implementation in order to meet time-critical requirements imposed by PHY/MAC processing. TRUMP provides a feasible solution to the CRM architecture and a rich set of tools for developing complex protocols including a meta-language compiler, wiring engine

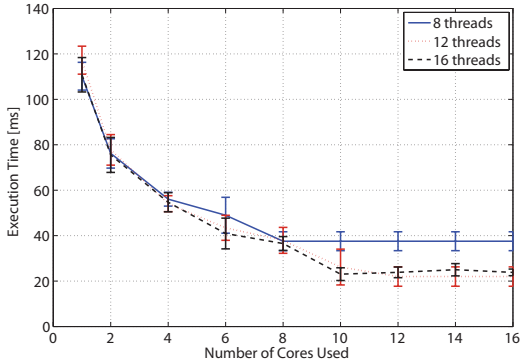

Fig. 12. Execution time of 16 independent tasks for computing mean and standard deviation over 10000 samples per task using different number of threads on different number of processing cores.

and a run-time optimizer. Our toolchain provides support for efficient protocol realization and execution on multi-core platforms which provide high computational power for high end communication and computing applications.

We have implemented TRUMP on various commercially available platforms such as WARP SDR platform, embedded Linux based PCs and resource constrained embedded sensor nodes. Our evaluation results on WARP and various sensor node platforms indicate that TRUMP is lightweight and highly compatible with embedded platforms. We have also shown that TRUMP enables fast run-time composition of protocols from a simple CSMA based MAC to a complex spectrum agile MAC [31]. By using decision making logic in the optimizer, run-time PHY/MAC protocol composition is observed to lead throughput gain up to 400 % which affirms the suitability of our toolchain for the cognitive radio network usage. Furthermore, evaluation results on multi-core PCs indicates that TRUMP is able to utilize multi-core computing fabrics for SDR platforms and results in more efficient execution of tasks. We are demonstrating TRUMP in the IEEE DySPAN'11 to showcase the capability of our toolchain.

We plan to further investigate TRUMP with parallel programming models on multi-core architecture to explore the possibility of parallel processing in alignment with the 2PARMA [26] approach. Different types of learning algorithms will be plugged in with TRUMP and be evaluated on WARP. The code for TRUMP will be publicly available to interested parties.

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 project ID: FP7-ICT-2009-4-248716 (2PARMA). We would also like to thank Marina Petrova for useful discussions, and anonymous reviewers for useful feedback.

REFERENCES

[1] Mitola J. III, "Cognitive Radio: An Integrated Agent Architecture for Software Defined Radio," *Ph.D. dissertation, Royal Institute of Technology (KTH), Sweden*, 2000.

[2] T. Yuecek and H. Arslan, "A survey of spetrum sensing algorithms for cognitive radio applications," *IEEE Communications Surveys & Tutorials*, vol. 11, no. 1, pp. 116–130, 2009.

[3] V. Ramakrishnan *et al.*, "Efficient and portable SDR waveform development: The Nucleus concept," in *Proc. of MILCOM'09*, Boston, MA, USA.

[4] M. Petrova, P. Mähönen, J. Riihijärvi and M. Wellens, "Cognitive Wireless Networks: Your Network Just Became a Teenager," in *Proc. of IEEE INFOCOM'06*, Barcelona, Spain.

[5] 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 Mob. Comput. Commun. Rev.*, vol. 12, no. 1, pp. 56–58, 2008.

[6] J. Ansari, T. Ang and P. Mähönen, "WiSpot - detecting Wi-Fi networks using IEEE 802.15.4 radios," in *Demonstration in EWSN'11*, Bonn, Germany.

[7] L.E. Doyle *et al*., "Experiences from the Iris Testbed in Dynamic Spectrum Access and Cognitive Radio Experimentation," in *Proc. of IEEE DySPAN'10*, Singapore.

[8] 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 IEEE DySPAN'05*, Baltimore, MD, USA.

[9] K. C. Huang, X. P. Jing and D. Raychaudhuri, "MAC Protocol Adaptation in Cognitive Radio Networks: An Experimental Study," in *Proc. of ICCCN'09*, San Francisco, CA, USA.

[10] D. G. Messerschmitt, "Rethinking components: From hardware and software to systems," *Proceedings of IEEE*, vol. 95, no. 7, pp. 1473–1496, 2007.

[11] 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.

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

[13] G. Nychis, T. Hottelier, Z. Yang, S. Seshan and P. Steenkiste, "Enabling MAC protocol implementations on software-defined radios," in *Proc. of NSDI'09*, Boston, MA, USA.

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

[15] M. Neufeld, J. Fifield, C. Doerr, and A. Sheth, "SoftMAC: A Flexible Wireless Research Platform," in *Proc. of HotNets'05*, College Park, MD, USA.

[16] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. M. Voelker, "Sora: High Performance Software Radio Using General Purpose Multi-core Processors," in *Proc. of NSDI'09*, Boston, MA, USA.

[17] A. Sharma and E. M. Belding, "FreeMAC: Framework for multi-channel mac development on 802.11 hardware," in *Proc. of PRESTO'08*, Seattle, WA, USA.

[18] M-H. Lu, P. Steenkiste and T. Chen, "FlexMAC: a wireless protocol development and evaluation platform based on commodity hardware," in *Proc. of ACM WiNTECH'08*, San Francisco, CA, USA.

[19] A. Sharma, M. Tiwari and H. Zheng, "MadMAC: Building a Reconfigurable Radio Testbed Using Commodity 802.11 Hardware," in *Proc. of IEEE SECON WSDR'06*, Reston, VA, USA.

[20] M. C. Ng, K. E. Fleming, M. Vutukuru, S. Gross, Arvind, and H. Balakrishnan, "Airblue: A System for Cross-Layer Wireless Protocol Development," in *Proc. of ACM/IEEE ANCS'10*, La Jolla, CA, USA.

[21] A. Dutta, D. Saha, D. Grunwald, and D. C. Sicker, "An architecture for software defined cognitive radio," in *Proc. of ACM/IEEE ANCS'10*, La Jolla, CA, USA.

[22] J. Ansari, X. Zhang, A. Achtzehn, M. Petrova and P. Mähönen, "Decomposable MAC Framework for Highly Flexible and Adaptable MAC Realizations," in *Proc. of IEEE DySPAN'10*, Singapore.

[23] J. Ansari, X. Zhang, A. Achtzehn, M. Petrova, and P. Mähönen, "A Flexible MAC Development Framework for Cognitive Radio Systems," in *Proc. of WCNC'11*, Cancun, Mexico.

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

[25] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. M. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, August 2000.

[26] C. Silvano *et al*., "2PARMA: Parallel Paradigms and Run-Time Management Techniques for Many-Core Architectures," in *Proc. of IEEE ISVLSI'10*, Kefalonia, Greece.

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

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

[29] J. Ansari, T. Ang and P. Mähönen, "Spectrum Agile Medium Access Control Protocol for Wireless Sensor Networks," in *Proc. of IEEE SECON'10*, Boston, MA, USA.

[30] H. N. Pham, Y. Zhang, P. E. Engelstad, T. Skeie, and F. Eliassen, "Optimal cooperative spectrum sensing in cognitive sensor networks," in *Proc. of IWCMC '09*, Leipzig, Germany.

[31] J. Ansari, X. Zhang and P. Mähönen, "A Decentralized MAC for Opportunistic Spectrum Access in Cognitive Wireless Networks," in *Proc. of ACM CoRoNet'10*, Illinois, Chicago, USA.

[32] J. Camp and E. Knightly, "Modulation rate adaptation in urban and vehicular environments: Cross-layer implementation and experimental evaluation," in *Proc. of MobiCom'08*, San Francisco CA, USA.