Wireless MAC Processors: Programming MAC Protocols on Commodity Hardware

I. Tinnirello[1], G. Bianchi[2], P. Gallo[1], D. Garlisi[1], F. Giuliano[1], F. Gringoli[3]
[1] CNIT / Università degli Studi di Palermo, Italy
[2] CNIT / Università degli Studi di Roma-Tor Vergata, Italy
[3] CNIT / Università degli Studi di Brescia, Italy

***Abstract*—Programmable wireless platforms aim at responding to the quest for wireless access flexibility and adaptability. This paper introduces the notion of *wireless MAC processors*. Instead of implementing a specific MAC protocol stack, Wireless MAC processors do support a set of Medium Access Control "commands" which can be run-time composed (programmed) through software-defined state machines, thus providing the desired MAC protocol operation. We clearly distinguish from related work in this area as, unlike other works which rely on dedicated DSPs or programmable hardware platforms, we experimentally prove the feasibility of the wireless MAC processor concept over ultra-cheap commodity WLAN hardware cards. Specifically, we re-flash the firmware of the commercial Broadcom AirForce54G off-the-shelf chipset, replacing its 802.11 WLAN MAC protocol implementation with our proposed extended state machine execution engine. We prove the flexibility of the proposed approach through three use-case implementation examples.**

***Keywords*-programmable MAC; WLAN 802.11, reconfigurability; cognitive radio**

I. Introduction

More than 20 years have elapsed since the establishment, in 1990, of the IEEE 802.11 Wireless Local Area Network committee. Initially foreseen as a technology for replacing Ethernet cables with wireless connectivity, IEEE 802.11 has been severely challenged by the highly heterogeneous needs emerged in the last two decades. Indeed, the original 802.11 CSMA/CA Medium Access Control (MAC) has shown significant shortcomings when facing the breakthrough rate improvements made available by the latest PHY enhancements (802.11n, 802.11ac), as well as when applied to scenarios and contexts such as ad hoc and mesh networks, vehicular environments, directional antennas, quality of service support, real time media streaming support, multi-channel operation, dynamic spectrum access, and many others.

Actually, the WLAN research community has found effective and ingenious solutions for adapting the 802.11 MAC operation to these new challenges. However, as for instance detailed in a comprehensive analysis carried out in the frame of the FLAVIA FP7 European project [1], most of the proposed MAC modifications do not comply with the 802.11 standard MAC operation. In the best case, i.e., when the required MAC amendments are endorsed by some 802.11 standardization task groups, several years may elapse before they become available in commercial cards/devices. More frequently, when the promoted MAC amendments are either deemed out of the standard task groups' scope, or mandate a "way too significant" departure from the native CSMA/CA MAC operation, their real world deployment is very unlikely, especially when they require changes in time-critical operations natively implemented in the network interface card.

This work has been carried out in the frame of the EU FP7-FLAVIA project, contract number 257263.

Programmable WLAN systems

A very first step to address the above concerns is to evolve from the current generation of closed network interface cards, implementing the very specific WLAN protocol stack, to programmable WLAN platforms[1], capable of permitting software-based modifications in the wireless access operation.

Obviously, the technical hurdle to face is how to support or modify time-critical operations which cannot be delegated to driver-level software modification, or controlled by dedicated overlay software modules running on the host computer.

The research community has mainly circumvented this issue by developing FPGA-based and/or Software Defined Radio platforms, such as [2], [3], [4], [5], [6], [7], [8], and has therein implemented and tested modifications to the wireless access operation. Quite recently, the public domain release of a (simplified) open firmware implementation [9] of the 802.11 Distributed Coordination Function, has further opened up the very appealing possibility to perform time-critical MAC modifications directly on commercial cards [10], [11], and thus with a much greater deployment potential.

From Wireless Cards to Wireless Processors

Despite the above discussed advances in programmable wireless systems, we share with a few other recent works [12], [13] the belief that wireless access programmability should go well beyond the ability to just "hack" firmware/software code implementing a pre-established MAC protocol stack, and should rather be *designed into* the MAC stack architecture.

In such a direction, this paper promotes the concept of *Wireless MAC processor*, a programmable device which i) provides a set of stateless Medium Access Control *commands*, and which ii) embeds a *MAC protocol engine* in charge of

[1]In this work, we concentrate on pure technical aspects, and we choose *not to discuss* the (controversial!) strategic implications behind wireless access programmability, such as vendors' business interest to pursue such a direction, possible fragmentation of the solutions' space, interoperability issues, etc.

executing a finite state machine able to *exploit* and *compose* the sequence of commands forming a desired protocol.

The Wireless MAC processor commands can be considered analogous to the instruction set of an ordinary CPU. They are meant to implement elementary *actions*, namely MAC operations such as transmit a frame, set a timer, freeze a backoff, etc, which may (or may not) be then executed in the appropriate sequence and/or under the occurrence of specific *events* and *conditions* mandated by a protocol logic. Going ahead with the same analogy, the MAC protocol engine can be somewhat related to an ordinary CPU control unit. It is in charge of executing an *user-developed* software program implementing the desired MAC protocol operation, which, in our proposed architecture, is provided in the form of an extended finite state machine.

Our contribution

The specific contributions of this paper include:

- the design of a MAC processor architecture;
- the design of its programming interface in terms of i) detailed identification of actions, events and conditions (for concreteness tailored to the specific WLAN case), and ii) their integration/handling through a programmable Extended Finite State Machine (XFSM);
- the implementation of the proposed system over a *commodity* card (the off-the-shelf Broadcom AirForce54G chipset), by replacing its 802.11 WLAN MAC firmware implementation with our MAC protocol engine;
- the proof-of-concept validation via three very diverse MAC extensions, tackling different MAC aspects (programmable frame replies, accurate scheduling of TX times, and fine-grained radio control).

Although we are the first to use the convenient and descriptive term *Wireless MAC processor*, we do *not* claim the novelty of the underlying MAC decomposition idea, which appears for instance in [12], [13]. Rather, besides the technical differences discussed in section II, we believe that our most distinguishing achievement is providing an architectural solution able to clearly decouple what the device is able to do (the pre-installed MAC commands) from what it is instructed - at run time - to do (the injected state machine).

II. Related Work

The ability to modify the operation of commodity WLAN systems goes along with the availability of public-domain open-source 802.11 MAC protocol code. Besides the significant expertise required to modify existing code, a further deployment barrier for many appealing MAC extensions consists in the limited extent to which software changes may affect the device operation. Indeed, early 802.11 devices were designed according to a *full-MAC* approach. The MAC layer was almost entirely implemented in the card hardware/firmware, and programmability of the relevant drivers (when provided as open source) involved a marginal set of functionalities. The flexibility of commodity WLAN cards has significantly improved since a number of vendors (including Intel, Ralink, Realtek, Atheros, Broadcom), started to exploit an innovative *soft-MAC* [15] design, transferring to the host processor non-time-critical MAC layer functionalities (figure 1).

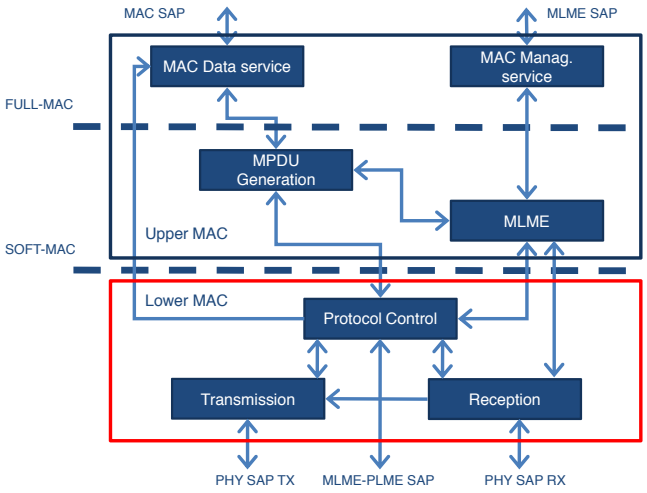

Fig. 1. WLAN MAC architectures: full-MAC vs soft-MAC.

Still, even in the *soft-MAC* case, the "Lower MAC", comprising crucial sub-systems such as transmission, reception and protocol control, remains hard-coded in the card. Although some chipsets (e.g. from Atheros and Broadcom) permit the tuning of selected MAC parameters (such as contention windows) via registers, more substantial MAC operation changes require access to the firmware code. To the best of our knowledge, no vendor has to date released an open source firmware, and the only available public-domain code is OpenFWWF [9], a recently released simplified DCF firmware implementation for Broadcom/AirForce54G chipsets. However, OpenFWWF extensions require reimplementation of large portions of assembly code, thus making it usable only by experts.

In parallel, a significant effort has been spent on the development of overlay software modules. Solutions such as the Overlay MAC Project [16], MultiMAC [17], FlexMAC [18], Soft-TDMAC [19], etc, do exploit firmware configuration registers and some driver hacks for building quite advanced MAC programming interface (for instance, MultiMAC permits to override the frame format, disable ACKs, RTS/CTS, virtual carrier sense, disable transmission backoff, etc). Even if notable implementations of custom MAC protocols, including TDMA-like ones, have been demonstrated, overlay approaches cannot get rid of some intrinsic limitations. Their scalability may be impaired by the need to overlap and duplicate similar functionalities at different layers; they remain constrained by the basic programming interface made available by the driver; and their limited ability to accurately control the card's timing prevents to deploy features such as programmable management of frame replies and handshakes, precise scheduling of medium access times, fine-grained radio tuning control, etc.

Clearly, the shift from commodity wireless cards to dedicated wireless platforms permits to push programmability much farther, although the beneficiaries remain mainly confined within the research community - real world deployment of costly and/or bulky platforms being unlikely.

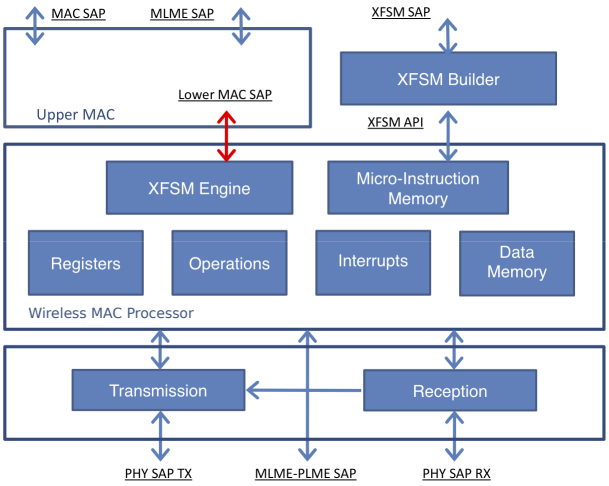

Fig. 2. Internal architecture of the Wireless MAC processor

Early platforms such as RUNIC [3] and CalRadio [2] re-implemented the 802.11 MAC protocol stack on, respectively, a Xilinx FPGA and a Texas DSP, interfaced to a commercial PHY-only Intersil 802.11b chip. As such, they permitted arbitrary MAC modifications, but protocol reconfiguration required a deep knowledge of the platforms and could only be done offline by recompiling the modified C code.

Software defined radio (SDR) platforms, such as GNURadio [4] and USRP [5], overcome the dependency on a specific PHY interface and permit to develop full-custom MAC/PHY cross-layer protocols. A large amount of work focuses on means to improve the slow SDR performance. On one side, solutions such as SORA [6] achieve a throughput comparable to commodity 802.11 hardware by distributing computation on multiple cores and by relying on sophisticated optimizations, as well as on an efficient radio control board. However, the software complexity makes protocol stack modifications not easy, as any update implies a redesign of the software block repartitions to multiple CPU cores. On the other side, platforms such as WARP [7] and AirBlue [8] improve performance by delegating most processing functions to the FPGA Hardware, meanwhile retaining the ability to closely control such functions via, e.g., registration of interrupt handlers, hardware triggers, read/write of hardware registers, etc. In the case of AirBlue, a modular organization coupled with careful design choices permits relatively easy modifications, changes in a module not affecting the others.

[12] and [13] are probably the works more closely related to the Wireless MAC Processor approach presented in this paper. Both start from a "breakdown" analysis devised to identify core MAC functions. Based on this, [12] proposes a *split-functionality* architecture, where time-critical MAC functions are run on the radio hardware, but their control is kept on the host PC. The architecture is implemented over the GNURadio and USRP SDR platforms. Conversely, in the *Decomposable MAC* framework proposed in [13] and detailed in [14] both basic blocks and protocol logic are supported on a WARP platform. The MAC logic is composed via a wiring engine that connects the basic blocks required to support the desired MAC operation.

Similar to [13], we also support the MAC protocol logic directly on the radio card. However, our work differs from [12], [13] for at least three major aspects. First, our breakdown analysis further includes events and conditions, in addition to MAC functions. Second, we leverage injection in the radio card (and more specifically in the designed MAC protocol engine) of Extended Finite State Machines, thus permitting a greater flexibility as well as run-time re-programmability of the MAC operation without interrupting the MAC service. Third, we provide an experimental proof of our architecture on resource-constrained commodity WLAN cards instead of powerful and capable FPGA radio boards.

III. Wireless MAC Processor Architecture

Our design starts from the consideration that most modern wireless cards do embed a general-purpose CPU for supporting the hardware control logic. We propose to push this approach farther, by transforming the card itself in a specialized processor, called *Wireless MAC Processor* (WMP). The WMP is devised to specifically handle hardware/PHY events and schedule actions on the hardware/PHY card resources, thus leaving the MAC protocol developer with the much simpler task of describing when and under which events and/or conditions such actions should occur. In other words, similarly to other processors specialized for handling digital signals (DSPs) or graphical images (GPUs), we introduce a processor specialized for handling MAC operations.

A. General WMP Architecture

The wireless MAC processor has been conceived as a CPU specialized for handling hardware/PHY events and actions by executing Extended Finite State Machines (XFSMs). We chose to abstract the definition of the medium access control logic in terms of state machine because they are very effective in modeling the behavior of sequential control operations, and most MAC protocols are formally described in terms of state machines. Figure 2 shows the internal architecture of the WMP, which includes five main components:

- an execution engine, running the provided XFSMs;
- a memory block for data and program;
- an interruption block passing the signals coming from the hardware to the execution engine;
- a set of operations which can be invoked by the execution engine, which include logic, arithmetic and flow control operations plus specialized MAC operations;
- a set of registers for saving system state parameters.

The figure also shows the interface towards the transmission/reception blocks and the PHY, and the interface towards the upper-MAC (external to the device, residing on the PC host). Finally, figure 2 shows a further module, called XFSM Builder. This is an optional module, external to the WMP, and running on the host PC, which, analogous to a compiler, permits the user to write state machines in an higher level (symbolic) language. The XFSM Builder further verifies that the underlying WMP is able to support (in terms of timers, data

XFSM formal notation		MAC engine meaning
S	symbolic states	MAC protocol states
I	input symbols	triggering *events*, e.g., hardware signals, timer expiration generated by the interrupt block, etc.
O	output symbols	MAC *actions*: commands acting on the hardware, performed by atomic functions either native in the device or implemented in the pre-loaded operations module (including arithmetic and logic operations, data creation and deletion, etc)
D	n-dimensional linear space $D_1 \times \cdots \times D_n$	all possible settings of n *configuration registers*
F	set of enabling functions $f_i : D \rightarrow \{0, 1\}$	set of *conditions* to be verified on the configuration registers for enabling the transitions
U	set of update functions $u_i : D \rightarrow D$	*configuration commands* devised to change the value of the configuration registers
T	transition relation $T : S \times F \times I \rightarrow S \times U \times O$	indicates the target state, the MAC commands and the configuration commands to be associated to each transition

TABLE I
MAC PROGRAMS EXPRESSED AS EXTENSIBLE FINITE STATE MACHINES

space, etc.) the high-level state machine provided, may further perform automatic state/transition optimizations, and may be equipped (as in our implementation described in section IV) with a graphical interface to further simplify the creation of MAC programs.

B. Execution Engine and MAC programs

The *execution engine*, analogous to the control unit of a microprocessor, is the core of the architecture. It performs the tasks of fetching the MAC program, translating it into logical operations and basic actions (as in RISC micro-instructions), scheduling actions on the hardware, and storing results.

The engine is in charge to execute the *MAC program*, written by the developer as an XFSM, and (dynamically) loaded in the WMP micro-instruction memory from the host PC. Starting from the current state, the engine waits for transition events (input signals). It then verifies whether optional triggering conditions are verified, in which case it executes the transition action and the state change.

XFSMs are a generalization of the finite state machine model and permit to conveniently control the *actions* performed by the MAC protocol as a consequence of the protocol logic, of *events* such as arrivals and timer expirations, and of *conditions* on configuration registers (whose settings can be verified for enabling state transitions and updated when the transition is triggered). Since the configuration memory is not explicitly represented in the state space, XFSMs allow to model complex protocols with relatively simple transitions and limited state space. Table I maps the formal definition of an XFSM, in terms of its abstract 7-tuple (S, I, O, D, F, U, T) [20], onto the specific terminology used in this paper, and the relevant meaning in terms of MAC primitives or parameters.

An user-defined MAC program is thus specified by the set of states S, the triggering conditions F and the transition relations T. The number of states and relations is in principle arbitrary

events	*actions*	*conditions*
CH_UP	set/get(reg, value)	dstaddr
CH_DOWN	switch_RX()	myaddr
RCV_ACK	TX_start()	queue_length
RCV_DATA	tx_ACK()	queue_type
RCV_RTS	tx_DATA()	cw
RCV_CTS	tx_RTS()	cwmin
END_BK	tx_CTS()	cwmax
COLLISION	switch_TX()	backoff
HEADER_END	prepare_header()	RTS_thr
MED_END	set_backoff()	ACK_on
MED_DATA_CONF	freeze_bk()	srcaddr
MED_DATA_START	resume_backoff()	frame_type
MED_DATA_END	update_cw()	fragment
QUEUE_OUT_UP	set_timer(value)	channel
QUEUE_IN_OVER	stop_timer()	tx_power
END_TIMER	more_frag()	plength
END_SIFS		

TABLE II
WMP APPLICATION PROGRAMMING INTERFACE: SUPPORTED EVENTS, ACTIONS AND CONDITIONS.

and depends on the device capability. Conversely, the set of *events* I, the set of *actions* O and U, and the set of registers D over which *conditions* may be enforced is *predefined* by the Wireless MAC Processor device and represent the *WMP programming interface*, detailed next. In other words, these sets represent the wireless device capabilities (for instance, the switch to a different frequency band) which, as such, cannot be programmed by the user, but must be supported by the device hardware, and can "only" be invoked and controlled by the user-defined state machine.

C. WMP Programming Interface

In order to define an interface covering most of the MAC programmability requirements emerged so far for WLAN systems, we analyzed several use cases [21] including a hybrid contention/polling, a TDMA-like, and a multi-channel access protocol. The set of identified events, actions and conditions able to support the analyzes use cases is summarized in Table II, and discussed in the reminder of this section. Obviously, since the WMP programming interface also represents what the device is able to do, it can be extended when new PHY capabilities (such as full duplex, beamforming, etc.), are available (similarly to the release of a more powerful PC processor).

Events. These are the set of signals either provided by the hardware interrupt block, or coming from the upper layers. The signals are generated by: i) the energy detection subsystem (CH_UP and CH_DOWN signals, i.e. start and end of channel busy intervals); ii) the receiver subsystem, (RCV_ACK, RCV_DATA, RCV_RTS, RCV_CTS, HEADER_END signals, i.e. end of reception of different frame types or frame portions; MED_DATA_START and MED_DATA_END/MED_END signals, which delimit the reception of a generic frame); iii) the frame control subsystem (COLLISION signal, i.e., checksum failure); iv) the transmitter sub-system (MED_DATA_CONFIRM signal, i.e., end of a frame transmission; v) the transmission and reception queues (QUEUE_OUT_UP and QUEUE_IN_OVER signals, respectively enqueuing of a new frame and overflow at the reception

queue); vi) the clock (END_BK, END_TIMER signal when a pre-set timer expires).

Actions. In addition to arithmetic, logic and control flow primitives, the operation block supports MAC-specialized operations, categorized into configuration commands and hardware commands. The former work on the WMP registers storing the information about the configuration of PHY and MAC parameters, which refer to: i) the energy detection mechanism: set/get(sensitivity), set/get(detection mode); ii) the transceiver: set/get(channel), set/get(power); iii) the head-of-line frame: update_retry(), more_frag(), prepare_header(); iv) the contention parameters: set/get(cwmin), set/get(cwmax), set/get(cw), set/get(RTS_thr). The second group of operations drive different card sub-systems: i) the transceiver subsystem: switch_RX(), tx_ACK(), tx_beacon(), tx_data(), tx_RTS(), tx_CTS(), switch_TX(), enable_ACK(); ii) the timers: set_bk(), freeze_bk(), set_timer(value); iii) the upper-layer interface: report().

Conditions. The WMP contains registers explicitly updated by WMP actions and/or implicitly updated by WMP events, which store information on the card configuration and network state. These registers include: the station MAC address and the queue registers (queue length/type), the transceiver registers (channel and power), the contention registers (timers, contention windows and backoff counter), the handshake registers, the frame registers (frame type, destination and source address, fragment), the medium state register. An example of register updated by a WMP action is the backoff counter register (set by invoking the set_bk() command), while an example of register automatically updated by hardware events is the medium state register (busy when a CW_UP event occurs).

IV. Implementation on a commodity WLAN card

To prove the viability of Wireless MAC processors, we challenged its implementation over an ultra-cheap commodity WLAN network interface card. Basically, for supporting the WMP paradigm, the card has to expose the WMP interface and run a MAC Engine.

We worked on the AirForce54G chipset from Broadcom [22], since one author of this paper contributed to develop the relevant open source firmware [9], and some documentation on the internal card structure and its general purpose processor, registers, timers and transmission/reception primitives is available. We replace the original card firmware with an assembly code implementing the WMP state machine execution engine, and map the previously described WMP programming interface into *actual* signals, operations and registers of the card[2]. Finally, for supporting the upper-MAC operations and interacting with the other protocol layers, we use the *b43* [23] soft-MAC driver, which adapts the Linux internal *mac80211* [24] interface to network card.

[2]Note that a similar firmware update and core procedure implementations can be in principle performed on a card based on a different chipset, provided that the chipset assembling tools and the documentation on the internal card structure are available.

A. Hardware Platform

The AirForce54G chipset is built around an 8 MHz processor with 64 registers supporting arithmetic, binary, logic and flow control operations. The other main blocks include:

- *TX and RX engine*. These blocks implement the transmission and reception actions of the WMP architecture. They encode and decode packets from internal representation to the 802.11b/g CCK and OFDM encodings; compute and verify the Frame Check Sequence; transmit and receive frames. Packet reception is performed by the RX engine in parallel to other processor tasks.
- *TX and RX FIFO queues*. These queues are interfaced to the host kernel. On the transmission path, packets forwarded from the driver are enqueued in the TX queue, from which the chipset pull frames and moves them into the TX engine. On the opposite path, the processor waits for a packet received by the RX engine, and pushes (or drops) the received data towards the host kernel.
- *Shared memory*. This memory space of 4 KB can be accessed also by the host and can be used for implementing the micro-instruction memory, i.e. the MAC program.
- *Internal code memory*. This 32 KB memory is used for implementing the MAC actions not natively supported by the hardware and the MAC engine.
- *Template RAM*. The RAM memory can be used for composing arbitrary frames (including customized frame replies) that can be pushed to the TX engine as if they came from the TX queue.
- *Internal registers and external conditions (EC)*. The internal registers keep hardware configuration settings. They can be set by the processor in response to changes in the EC to program the radio interface and set up timers.

B. WMP Implementation choices and issues

Our actual XFSM execution engine implementation works as summarized in figure 3-(a). The main difference with the WMP conceptual architecture is the management of events. Hardware signals are in fact directly handled by registers and cannot be asynchronously intercepted by the MAC engine implemented in the firmware. For example, the end of a frame reception (i.e., the MED_DATA_END event of the WMP) is signaled by a change of status of a specific frame reception register. Hence, we were forced to resort to a cyclic polling of the event-logging registers. To limit polling delay, per each MAC program state, we restricted the list of polled registers to those logging events which could eventually trigger non-null state transitions. As shown in figure 3-(a), the relevant list is pre-loaded at each state transition.

From the above discussion, it follows that our implementation does not strictly distinguish between events and conditions, both of them being verified by monitoring card registers. However, the condition verification does not require a cyclic polling, being performed only when a transition is triggered. For what concerns the WMP programming interface implementation, all the register configuration actions and part

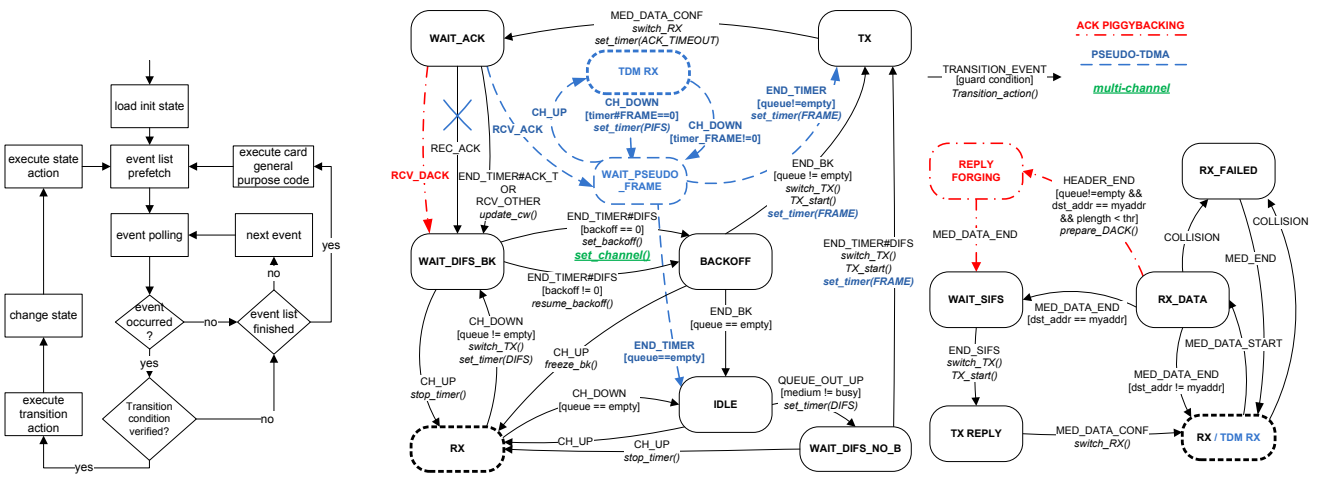

Fig. 3. MAC Engine work flow (a) and MAC program examples (b).

of the MAC actions summarized in table II were natively supported by the chipset; the remaining ones were developed and pre-installed as micro-code procedures. Finally, we mapped all the table II's events and conditions into the Airforce54G internal registers.

C. WMP Machine language

To permit the MAC engine to execute an XFMS, the latter must be coded in a suitable machine language, analogous to a bytecode. Let n_s be the number of symbolic states, and let n_e be the number of input events in I, the easiest approach is to code the XFSM as an $n_s \times n_e$ table. At each location (i, j), the table stores the state transition when event j is received at state i. Each transition has been defined by means of the 6-bytes triplet (a, c, s), where:

- $a \in O + U$ is a 2-bytes MAC *transition* action, where the first byte identifies the action label, and the second byte the action parameter (needed in case of configuration actions);
- $c \in F$ is a 2-bytes condition enabling the transition (first byte = register name, second byte = register state);
- s is the target state, coded with 2 bytes.

Note that we did not limit to 1 byte per state as the number of actual states may become larger than the nominal ones: when multiple actions/conditions are associated to a same transition, as a consequence of the above coding, the state must be split into a sequence of intermediate states, each triggering at most one action and verifying at most one condition.

In practice, to cope with the severe memory limitations of the chipset (only 4 KB are available for storing the MAC program table), we optimized the memory occupancy by replacing each table's row with a list containing only the non-null state transitions. As each state generally reacts to a number of input events much lower than the total inputs number (i.e., the table is sparse), skipping null-transitions significantly reduces the required memory space. Moreover, as a second optimization, we enabled the possibility to use the second byte of the state labels for encoding an additional *state* action (with no parameter) to be executed after the state transition.

D. XFSM builder

To avoid writing MAC programs in the above described machine language, we developed an XFSM builder which includes a graphic XFSM editor on the eclipse platform for composing MAC program, and a "bytecode" compiler which translates an XFSM graphical representation into the machine language understandable by the firmware's MAC engine. The bytecode can be uploaded to the device by using debug tools, or can be injected from the host to the card by forwarding special packets whose payload carries the MAC bytecode. Loading a new bytecode on the chipset allows changing on-the-fly the card behavior without any re-compiling operation.

V. Functional and Performance Evaluation

In order to validate the proposed approach, we first verified that our WMP implementation on the Broadcom card could efficiently support a full "Lower MAC" protocol implemented in the WMP XFSM machine language instead of micro-code. The obvious choice was to (re)implement the legacy 802.11 DCF as an XFSM executed by the WMP, and compare its performance with the benchmark provided by the native Broadcom's firmware, as well as with the performance provided by the OpenFWWF firmware (i.e., DCF as well reprogrammed on the card, but via straight firmware re-coding).

The relevant XFSM is illustrated in figure 3-(b) (black states and transitions - the same figure shows, with different colors, the extra transitions and states modeling the extensions discussed next). Besides the self-explaining state labels, input events, and transition arrows, the figure reports guard conditions and actions (when associated to a transition) in square brackets and in italic style, respectively. In case of multiple timers simultaneously active, the figure also specifies the source register (e.g. #DIFS) of the end timer events. For graphical convenience, the figure separates the TX state machine (left) from the RX one (right). Our performance tests,

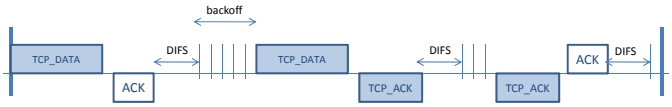

Fig. 4. Example of Piggybacked ACK operations.

performed for all the supported PHY rates (up to 54 Mbps) did not show any noticeable difference with respect to the Broadcom and OpenFWWF benchmarks.

We then ran a second set of experiments, having the *functional* goal of showing that the availability of wireless MAC processors may permit very easy and fast modifications in the Lower MAC operation. To this purpose, in the following subsections, we show how three MAC modification examples are readily deployed via straightforward state machine modifications. We remark that some selected examples are simplistic, and do *not* aim at being scientific proposals. Rather, they are chosen because they are very easily explained in the limited space available, and, most important, because they tackle distinct MAC operation aspects which indeed recur in several literature proposals: programmable management of frame replies, (section V-A), precise scheduling of the medium access times (section V-B) and fine-grained control of the radio channels (section V-C). Note that without the WMP, despite their simplicity, an actual (accurate) implementation of such examples likely requires complex firmware-level hacks.

For monitoring the behavior of the MAC program executions, apart from measuring the throughput performance, we also used a customized tool for acquiring and processing channel activity traces. The trace acquisition is based on USRP [5], while the trace processing is performed in MATLAB for deriving the power levels of the channel samples[3].

A. PiggyBacked ACK

As an example of access protocol using an acknowledgment mechanism different from the standard one, we considered a PCF-like frame exchange under random access. The basic idea is very simple: when a given station wins the medium contention and transmits its DATA frame on the shared channel, if the destination station has a not-empty transmission queue and the frame size is not longer than a given threshold, it can reply with a DATA+ACK frame (a standard data frame, whose subtype field is 0001 rather than 0000), i.e. with a data frame carrying in piggybacking an acknowledgment for the received frame. The enabling threshold is used for limiting the DATA+ACK transmission time, since such a frame transmission is not protected by the duration field specified in the previous DATA frame and could be corrupted by hidden nodes. The DATA+ACK frame is sent after a Short Interframe Space (SIFS) interval from the end of the DATA frame reception, as in the case of normal ACK frames. If the destination station has an empty transmission queue or a too long frame, the normal ACK frame is sent as usual.

[3]We chose to use a SDR platform rather than developing a monitoring XFSM for validating the performance of the MAC program execution by means of a third-party instrument.

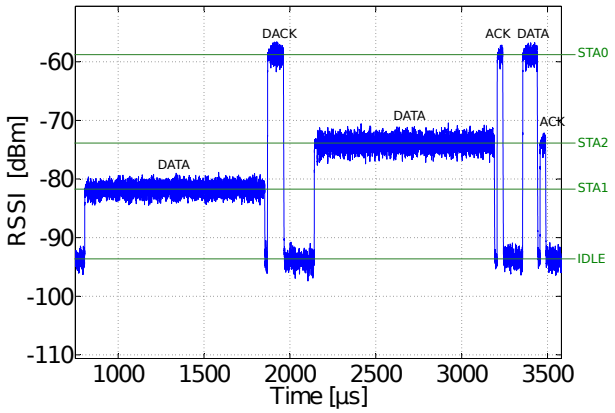

Fig. 5. An experimental trace of medium occupancy times under legacy DCF and Piggybacked ACK.

Figure 3-(b) shows how such a variant can be easily defined in terms of an update (red states and transitions) on the DCF finite state machine. Let us consider the receiver subsystem, which includes the main modifications. Starting from an ongoing reception, i.e. from the RX_DATA state, when the header reception is completed (HEADER_END event), the station transits to the REPLY_FORGING state under the condition $[dst\_addr = myaddr]$, $[pklength < thr]$ and $[queue! = empty]$. While in this state, the station continues the reception process and simultaneously prepares the DATA+ACK frame reply (labeled as DACK). If the transmission queue is empty or the packet size is higher than the threshold (but the frame destination is the target station), the receiver sub-system transits to the normal WAIT_SIFS state at the reception end (MED_DATA_END event).

Figure 4 illustrates an example of channel access operations of two stations involved in a TCP data session, under the piggybacked ACK scheme. In [25], where the application was originally introduced, a performance comparison between the piggybacked ACK and the standard DCF protocol shows that the piggybacked ACK scheme leads to a TCP throughput gain of about 20% for the considered scenario.

Experimental Results: We implemented the state machine of the piggybacked ACK (with a piggybacking threshold set to 200 byte) by porting the implementation discussed in [25] on the engine-based firmware platform. We set up a simple networking scenario: two TCP data sessions are originated between two different senders (STA1 and STA2) and a common receiver (STA0). The receiver uses two data transport services: the piggybacked ACK for the TCP sink connected to STA1, and the legacy DCF for the TCP sink connected to STA2. The TCP sessions are created by the *iperf* software with a payload size of 1500 byte, while all the stations use a fixed 802.11g rate equal to 12 Mbps.

Figure 5 shows a segment of the channel activity trace acquired (after a transient phase) by the USRP, in terms of received power levels. Because of the different distance between the stations and the USRP, the received power levels permit to distinguish the three transmitting stations. The figure includes three channel access handshakes, where normal ACKs and

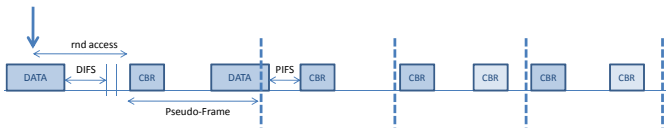

Fig. 6. Example of Pseudo-TDMA operations.

TCP acks are characterized by a different duration (namely, about 93 μs for TCP acks and 31 μs for normal ACKs). In the first handshake, STA1 sends a TCP segment which is followed by a TCP ack sent by STA0. In the second case, STA2 sends its data segment and STA0 answers with a standard ACK. Finally, the TCP ack sent by STA0 to STA2 contends to the medium as a normal data packet and is acknowledged by a normal ACK frame (sent by STA2).

B. Pseudo-TDMA

In order to prove that our platform can provide a precise scheduling of medium access times, we considered some simple DCF extensions devised to support a pseudo-TDMA mechanism, similar to the one described in [26]. The scheme assumes that a preliminary admission control test has to be verified before contending for a pseudo-slot. A graphical representation of the pseudo-frame organization is given in figure 6, where for space reasons the ACK frames are not explicitly shown and have to be considered included in the transmission boxes. After a successful random access, the admitted CBR flow schedules the next medium access at the end of a fixed time interval, corresponding to the pseudo-frame duration. If the medium is idle, the transmission is immediately performed when the timer expires. Otherwise, the channel access can be performed at a very high priority after a PIFS time from the end of the previous transmission. Subsequent CBR flows admitted in the network will start the first random access during the empty pseudo-frame interval, thus obtaining a different pseudo-slot position within the frame. In case of transmission failure, a new random access has to be performed for gaining a new pseudo-slot.

The implementation of such a scheme in terms of updates on the standard DCF finite state machine is shown in figure 3-(b) with blue states and transitions. After the first ACK reception, the transition to the TX state can be performed from

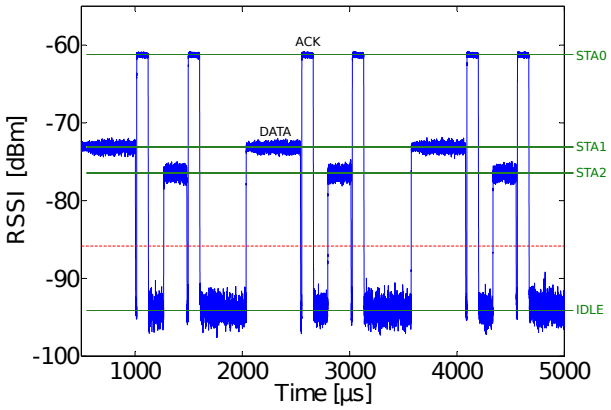

Fig. 7. An experimental trace of medium access times under Pseudo-TDMA.

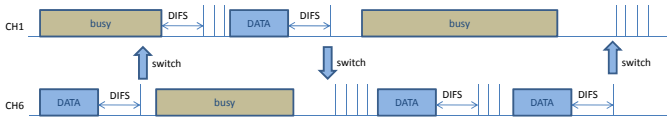

Fig. 8. Example of randomized multi-channel MAC operations.

the WAIT_PSEUDO_FRAME state at the expiration of the frame timer. If during the timer expiration a CH_UP signal is revealed, the machine transits to TDM_RX. When the medium is idle again (CH_DOWN event), a PIFS timer is set in case of expiration of the pseudo frame interval. In case of empty queue, the machine comes back to the IDLE state.

Experimental Results: Figure 7 shows an example of channel activity trace acquired by the USRP during an experiment in which two stations executing the Pseudo-TDMA program send packets of different size (namely, 500-byte packets from the first station STA1, and 100-byte packets from the second STA2) towards a common destination station (STA0). Both the stations employ an 802.11b PHY with a transmission rate set to 11 Mbps and a pseudo-frame interval of 1.536 ms (i.e. 0x600 μs). In the figure, the different frame lengths allow to distinguish the two transmitters, while the different power levels allow identifying acknowledgment transmissions and idle times. The medium access times are scheduled with a precision (of the order of micro-seconds) not achievable with driver level hackings.

C. Randomized multi-channel access

As an example of fine-grained radio control, we chose the possibility to perform a per-frame dynamic configuration of the radio in terms of transmission channel tuning. Also in this case, we focused on the analysis of the programmability requirements for supporting the access scheme, rather than on performance optimization. We considered a simple access mechanism for single radio nodes able to switch between two transmission channels and transmitting traffic towards a multi-radio access point (simultaneously synchronized on both the channels). At each backoff extraction, the nodes extract a new backoff counter and randomly choose to change or not the transmission channel.

The updates on the legacy DCF state machine are included in figure 3-(b) in terms of a new action (in green) associated to the transition from state WAIT_DIFS_BK to state BACKOFF. Although the protocol is a very simple example of multi-channel MAC, it may lead to the potential benefits depicted in figure 8. In case of independently interfered channels, the hopping mechanism allows to partially aggregate the bandwidth available on each channel. In practical, the channel switching operation could not be instantaneous. Therefore, more efficient solutions could be implemented by enabling the channel switching on the basis of traffic-load estimators (rather than being random), in order to reduce the channel wastes occurring at each switch.

Experimental Results: In this experiment we used an 802.11g PHY, with the data rate set to 18 Mbps. We considered a multi-channel MAC program, switching between channel

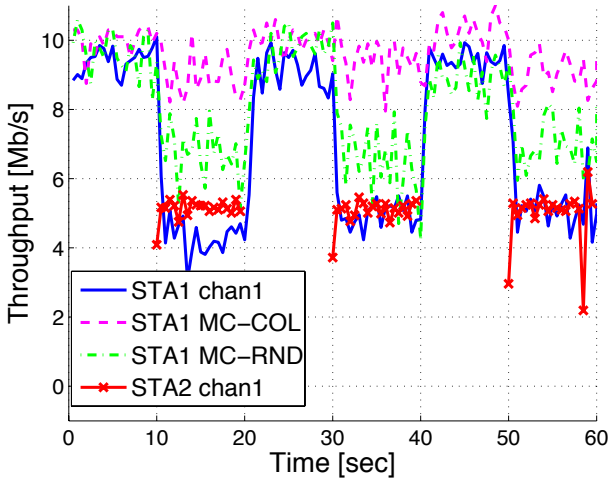

Fig. 9. Throughput performance of a station randomly hopping between an idle channel and a bursty interfered channel.

1 and channel 10 with a switching probability set to 1/16, and a DCF legacy program working on channel 10. For quantifying the performance of the scheme and the channel switching overheads, we compared the throughput perceived by a station loading the multi-channel MAC program (dashed green line) or the DCF legacy program (blue line), when it contends with another legacy station which transmits intermittently on channel 10 (as shown by the red curve in figure 9). The source rate of the contending station is set to 5 Mbps. When the contending station is not active (i.e. in the time intervals [0,10]s, [20, 30]s and [40, 50]s) there is no remarkable performance difference between the legacy DCF scheme and the multi-channel scheme, thus proving that the channel switching overhead is almost negligible. However, when channel 10 is interfered, the multi-channel MAC allows to increase the average throughput from 5 Mbps up to about 7 Mbps (i.e. to about the average value between 10 Mbps gained on channel 1 and 5 Mbps gained on channel 10). By activating the channel switching condition only in case of collision (with a 0.5 switching probability), the performance of the multi-channel MAC (dashed purple line) can be further improved, by maintaining almost a fixed 10 Mbps throughput.

VI. Conclusions

Current network interface cards are meant to implement a specific MAC protocol stack, and can be reprogrammed only having access to the (very complex, and rarely public domain available) firmware code.

The Wireless MAC processor introduced in this paper is a new approach to conceive wireless access devices. Rather than supporting a *given* protocol stack, our Wireless MAC processor supports MAC *commands* and relevant triggering events and conditions, which are then composed and controlled by a *MAC protocol engine*, namely an extended finite state machine executor. Thus, an eventually full-custom MAC protocol can be programmed by simply injecting (or replacing at run-time, for dynamic MAC protocol reconfiguration) in the Wireless MAC Processor an *user-defined* state machine modeling the desired MAC operation.

In this paper, we have provided an architecture design for the Wireless MAC Processor, we have specified its programming interface in terms of *actions*, *events* and *conditions*, and we have concretely proved that it can be implemented over an ultra-cheap commodity WLAN card. Three very diverse use-case examples have been finally implemented and validated, to show the flexibility and versatility of the proposed approach.

References

[1] FP7 ICT FLAVIA project, "Analysis of state of the art: contention-based extensions", *Deliverable D6.1*, Oct. 2010; http://www.ict-flavia.eu
[2] http://calradio.calit2.net/calradio1.htm
[3] A. Di Stefano, G. Terrazzino, C. Giaconia, "FPGA Implementation of a Reconfigurable 802.11 Medium Access Control", Int. Conf. on Wireless Reconfigurable Terminals and Platforms (WIRTEP), Rome, April 2006.
[4] The GNURadio Software Radio, http://gnuradio.org/trac
[5] USRP. The universal software radio peripheral, http://www.ettus.com
[6] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, G. M. Voelker, "Sora: High Performance Software Radio Using General Purpose Multi-core Processors", NSDI 2009.
[7] http://warp.rice.edu/trac
[8] M. C. Ng, K. E. Fleming, M. Vutukuru, S. Gross, Arvind, H. Balakrishnan, "Airblue: A System for Cross-Layer Wireless Protocol Development", ACM/IEEE Symp. on Architectures for Networking and Communications Systems (ANCS) 2010.
[9] Open firmware for WiFi networks, http://www.ing.unibs.it/openfwwf/
[10] B. Han, A. Schulman, F. Gringoli, N. Spring, B. Bhattacharjee, L. Nava, L. Ji, S. Lee, R. Miller, "Maranello: Practical Partial Packet Recovery for 802.11", USENIX NSDI'10, Apr. 2010.
[11] B. Han, F. Gringoli, L. Cominardi, "Bologna: Block-Based 802.11 Transmission Recovery" 2nd Wireless workshop of the Students, co-located with ACM MobiCom, Sep. 2010.
[12] G. Nychis, T. Hottelier, Z. Yang, S. Seshan, P. Steenkiste, "Enabling MAC Protocol Implementations on Software-defined Radios", NSDI'09, 2009.
[13] J. Ansari, X. Zhang, A. Achtzehn, M. Petrova, P. Mahonen, "Decomposable MAC Framework for Highly Flexible and Adaptable MAC Realizations" Proc. of IEEE DySPAN 2010, April 2010, pp.1-2.
[14] J. Ansari, X. Zhang, A. Achtzehn, M. Petrova and P. Mahonen "A Flexible MAC Development Framework for Cognitive Radio Systems" Proc. of the IEEE WCNC, Cancun, Mexico 2011.
[15] M. Neufeld, J. Fifield, C. Doerr, A. Sheth, D. Grunwald, "SoftMAC - Flexible Wireless Research Platform" HotNets, Nov. 2005.
[16] http://www.cs.berkeley.edu/ ananthar/oml.html
[17] C. Doerr, M. Neufeld, J. Fifield, T. Weingart, D.C. Sicker, D. Grunwald, "MultiMAC - An adaptive MAC Framework for Dynamic Radio Networking", IEEE DySPAN 2005, Nov. 2005.
[18] M.H. Lu, P. Steenkiste, and T. Chen, "Using commodity hardware platform to develop and evaluate CSMA protocols", ACM WiNTECH 2008, Sep. 2008, pp. 73-80.
[19] P. Djukic, P. Mohapatra, "Soft-TDMAC: A Software-Based 802.11 Overlay TDMA MAC with Microseconds Synchronization", IEEE Trans. on Mobile Computing, Issue 99, April 2011,
[20] K. T. Cheng, A. S. Krishnakumar, "Automatic Functional Test Generation Using The Extended Finite State Machine Model", ACM Int. Design Automation Conference (DAC), 1993, pp. 86-91.
[21] FP7 ICT FLAVIA project, "Report on scenarios, services and requirements", *Deliverable D2.1.1*, Jan. 2011, http://www.ict-flavia.eu
[22] http://www.broadcom.com/products/brands/AirForce
[23] http: //linuxwireless.org/en/users/Drivers/b43.
[24] Linux kernel mac80211 framework for wireless device drivers. http://linuxwireless.org/en/developers/Documentation/mac80211.
[25] P. Gallo, F. Gringoli, I. Tinnirello, "On the Flexibility of the IEEE 802.11 Technology: Challenges and Directions", Future Networks and Mobile Summit, June 2011.
[26] G.S. Paschos, I. Papapanagiotou, S.A. Kotsopoulos, K. Karagiannidis, "A New MAC Protocol with Pseudo-TDMA Behavior for Supporting Quality of Service in 802.11 Wireless LANs", EURASIP J. on Wireless Commun. and Networking, Vol. 2006, pp. 1-9.