Virtual Machine for Software Defined Radio: Evaluating the Software VM Approach

Riadh Ben Abdallah, Tanguy Risset and Antoine Fraboulet
Citi, Insa-Lyon
6, avenue des Arts
69621 Villeurbanne Cedex, France
Emails: {riadh.ben-abdallah, tanguy.risset, antoine.fraboulet}@insa-lyon.fr

Jérôme Martin
CEA-LETI, MINATEC,
17, rue des Martyrs,
38054 Grenoble Cedex, France
Email: jerome.martin@cea.fr

***Abstract*—We study the impact of using a virtual machine for the configuration of radio physical layer protocols on a real hardware platform: the Magali chip. The virtual machine is programmed in software on the ARM processor present on the platform. We evaluate the additional cost of the virtual machine layer on the effective implementation of telecommunication physical layer protocols. The results, obtained using the mixed SystemC/VHDL cycle accurate simulator of the Magali platform, show that, although the proof of concept is valid and functional, extra optimizations, such as additionnal hardware mechanisms, will be necessary to obtain real-time performance.**

I. Introduction

Software defined radio is now foreseen as the next technological shift that will drive commercial success for new mobile embedded systems. Automatic and dynamic adaptation to the strongest (or cheapest) radio protocol as well as global minimization of energy consumption over a set of mobile nodes may be reached only with the availability of software reconfiguration of the protocol physical layer.

Following the pioneering work of Mitola [1], software defined radio (SDR) has been "de facto" defined as the ability to *program* the physical layer of the radio protocol used for wireless communication. This does not mean, of course, that the protocol is fully realized in software. High bandwidth telecommunication protocols require hardware components for a real-time implementation. However, the same hardware components (e.g., a FFT component) are used in several different protocols with different parameters. What must be done in software is the *configuration* and *control* of these hardware components.

As soon as a SDR platform is programmable, its programs (sometimes called *waveform programs*) should be *easy* to develop and *reusable*. The reasons for that are that $i)$ programmability very quickly leads to intractable complexity and time-consuming debugging process and $ii)$ the vast quantity of hardware mobile platforms makes it impossible to develop one waveform program per hardware platform. This is the basic motivation for the use of a virtual machine for SDR: a dedicated virtual machine available on each hardware platform would be a solution to the "program once run everywhere" ideal scheme for waveform programs.

This Radio Virtual Machine concept (RVM) has been proposed in various works [2], [3], we have presented in [4] our proposal that includes a specific mechanism for the virtual machine to act on the signal processing data-stream. An important question-mark remains over the overhead induced by the use of a virtual machine for waveform configuration. This overhead can be measured in terms of additional hardware complexity and/or additional software complexity which itself can be divided into run-time performance and compile time issues (e.g., memory used). We propose in the paper to evaluate the practical impact of the use of a virtual machine on an existing platform: the Leti Magali chip [5].

In this work, we investigate a proof of concept to implement a *software* RVM: we reuse an existing SDR platform without any additional dedicated hardware. Given an existing hardware platform able to execute different waveform programs, such as Magali, we provide answers to the following questions: how much does it cost to add a software virtual machine on this platform? Will the obtained RVM respect real timing constraints of 3G telecommunication protocols? Our experiments have been done with a port of the Lua virtual machine on the ARM processor present on Magali. These experiments show that our software RVM implementation is approximately 2 to 6 times slower than native implementation which tends to prove that the overhead introduced can be managed and that more optimized VM should be used and investigated.

The paper is organized as follows: an overview of existing SDR platforms is presented in section II. Our RVM proposal is then rapidly recalled in section III. Section IV introduces the Magali platform. Implementation choices are then detailed in section V, and section VI describes the experimental tests realized in order to evaluate our RVM implementation. Finally, section VII presents our conclusions.

II. Existing SDR Platforms

SDR offers faster time to market and shortens development cycle of new products. Due to these economic issues, SDR technologies have shown quick advancements during the last few years. An important number of SDR platforms with various architectures have been proposed both by academic research laboratories and by commercial companies. In the following, we briefly present a representative sampling of SDR platforms.

A. DSP-*centric platforms*

SDR platforms allowing to implement full-software signal processing modules are highly flexible. Many companies (Sandbridge[6], picoChip, Fujitsu, Icera, Infineon, NXP, etc.) propose DSP-centric integrated circuits for SDR. Some typical examples are:

- picoArray [7] processor from picoChip: This chip has an architecture which integrates hundreds of small DSPs. PicoArray can be programmed in ANSI C within a dedicated development environment. It offers a high performance of 200GIPS and 30GMAC/s at 160MHz. Associated with some accelerator IPs (FFT, Turbocodes, etc.) it is capable of implementing a complete software-defined WCDMA modem.
- X-GOLD^{TM}SDR 20 from Infineon technologies is a programmable baseband processor for Multi-Standard Cell Phones. Infineon provides a hardware/software solution that permits to support GSM, GPRS, EDGE, W-CDMA, HSDPA, HSUPA, LTE, 802.11a/b/g/n, DVB-T/H protocols. This platform also includes hardware accelerators for resource consuming computations.
- EVP(Embedded Vector Processor) [8] from NXP (owned by ST-Ericsson): this architecture is able to support multimode LTE with full compliance to the current draft of revision 8 of the 3GPP standard.

B. *Heterogeneous Platforms*

These platforms are mainly composed of dedicated hardware processing units. At least one CPU is usually required for the platform to control hardware operators. Experimental platforms contain one or multiple FPGAs in order to make possible the implementation of newly designed algorithms. To a lesser extent, DSPs are used to implement standard specific functionalities which doesn't require high computational performance. Here are some example of heterogeneous platforms:

- Small Form Factor (SFF) SDR Development Platform from Lyrtech: it embeds one DSP and one FPGA for baseband processing, connected to a RF front-end. A dedicated development environment is also provided.
- Universal Software Radio Peripheral (USRP): is a hardware platform designed for the GNU Radio project [9]. This platform have to be connected to a PC platform through USB interface (Control and I/Q samples transfer).
- Kansas University Agile Radio (KUAR) platform [10]: is a low cost experimental SDR platform including a 1.4GHz Pentium M with 1GB RAM and a Xilinx Virtex2 FPGA. A gigabit Ethernet and PCI-express links are provided for connection to host computer.

It is worth mentioning that an important number of DSP-centric platforms can be found in the literature. They offer high flexibility but are limited in terms of performance (computation speed and energy consumption) compared to platforms with dedicated hardware accelerators. We also notice that a centralized CPU is always present for platform control.

III. Radio Virtual Machine Proposal

The execution model we propose for our virtual machine implementation is meant to deal with configuration and control of a SDR platform. This is an abstraction of the architecture on which will run the streaming computation needed for the SDR platform. This model has been previously described in [4], and its adequacy with the description of telecommunication waveforms has been shown. This section sums up its main characteristics.

In our case, the execution model is a set of IPs interconnected by an efficient communication mechanism. These IPs can be software or hardware blocks (we detail this below), they are reconfigurable and can accept runtime parameters. On behalf of the processing IPs that will take part of the radio protocol stream processing, one of these IPs must implement a particular controller (in our case it is the virtual machine) and can be a dedicated component or implemented as general purpose processor as it is the case in the present prototype.

The platform reconfiguration management using a virtual machine can preserve genericity of implantation until the last moment: downloading the bytecode program on the device. Once the bytecode is downloaded to the target RVM several runtime steps still need to be done.

- Allocation and resource sharing can and should be associated with the virtual machine to increase the portability of the code and go through the mechanism of operators virtualization.
- Just in time compilation techniques can be used on the configuration bytecode to increase the virtual machine efficiency and make use of the the platform specific optimization opportunities.

These steps are not discussed here. In this paper, we place ourselves in the case where the allocation of resources may be made by simple association between the blocks available on the system composed of the hardware IP and the virtual machine process.

Calculations used in the computation stream can be instantiated by hardware or software. The hardware blocks are seen as hardware accelerators for the computations. The software blocks can be instantiated on general purpose processors, DSP or in our case *within the virtual machine itself*. This process is an important component for a configurable virtual machine for SDR platforms.

From a functional point of view, communication management among IPs is achieved using First-In-First-Out (FIFO) channels. These communications are either performed by the hardware IPs themselves or, in case of software IPs, by DMAs. In both cases, our model uses simple communication patterns (point-to-point, with a known number of data). These components configured by the virtual machine and interact with it through interrupts in addition to data transferred to and from memory blocks.

Our RVM does not make any strong assumptions about how the platform should implement the communications. However, data integrity during transfers between modules or blocks must

be guaranteed by the platform. The absence of communication deadlocks among blocks must be ensured at compile time or during bytecode validation into the VM. Once configured the IPs should be able to receive their data without intervention of the virtual machine.

IV. Proof of Concept Platform

This section gives an overview of the Magali chip, which has been used as proof-of-concept platform to evaluate the performances of our radio virtual machine. A synoptic diagram of the chip is shown on figure 1.

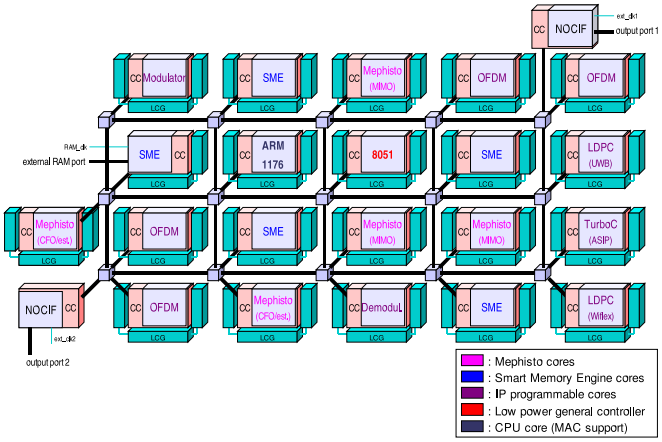

Fig. 1. Magali system-on-chip

Magali is a system-on-chip which targets the physical and MAC layer processing of advanced telecommunication applications such as 3GPP-LTE or IEEE802.16e (WiMax) [5]. It is based on a 2D-mesh asynchronous network-on-chip (NoC) [11] made of 5-port routers interconnected to each other by bidirectional links. Each computing unit is connected to one port of a router. There are several kinds of computing units in Magali: IPs, DSP, DMA, etc. We review them in next paragraph, in the rest of the paper we refer to a computing unit as a *component*. A magali *component* might be a hardware IP or (e.g. FFT) a software program on a DSP.

IP components are dedicated to a specific telecom computation (OFDM components which include FFT/iFFT, LDPC and Turbocodes coding/decoding blocks, bit processing blocks, etc.). They offer a certain degree of configurability to address the diversity of targeted applications (e.g., FFT size, OFDM frame format, etc.). The Smart Memory Engine (SME) [12] and the Mephisto blocks are specialized programmable components: the SME is a programmable DMA that is able to select and reorder data, whereas the Mephisto is a VLIW DSP used to address specific telecom functions that involve highly flexible computations (channel estimation, MIMO decoding, etc.). Finally an ARM1176JZFS processor core brings the programmability needed for high-level control and MAC processing. It is also responsible for configuring and controlling all other computation units in Magali.

In the Magali chip, every functional unit is connected to its router through a hardware block called Communication and Configuration Controller (CCC) [13], called CC on figure 1. The CCC is responsible for optimizing dataflow communication on the chip. Hence, programming an application on Magali consists in configuring these controllers so that they may start and control data input flows, core computation and data output flows for each component of the platform. For performances purpose, the CCC is able to manage complex configuration sequences where the component receives and sends data from/to several other components and processes different computations. This complex configuration sequences reduce the amount of work of the ARM core and prevents communication bottleneck at its NoC interface.

The choice of Magali as a test case for our radio virtual machine relies on several criteria. Firstly, it is a chip that addresses state-of-the-art telecommunication standards, with dedicated optimized IPs, and therefore it has sufficient computation performances and power efficiency to address realistic SDR challenges. Secondly, using such a heterogeneous platform is a way to evaluate our RVM programming model against several families of computing components with variable configurability and programmability features. If our model is able to deal with this chip it is obviously able to handle homogeneous platforms such as DSP-only ones. Another advantage of the chip is that the ARM processor makes it possible to prototype software calculations that have not been included in optimized IPs. This is useful to evaluate the possibility for the platform to cope with new emerging standards, which is a key concept of SDR. Finally, the design environment provided with Magali gathers all the necessary tools to efficiently prototype the RVM: embedded RTOS, ARM ISS, high level model in SystemC for quick simulation, and an evaluation board for experiments in real conditions.

V. Implementation Choices

The RVM concept is essentially a domain specific virtual machine. In order to have rapidly a RVM prototype we chose to not start from scratch. Instead, we have conducted a preliminary study on the existing VMs taking into account a set of considerations. Then we selected one suitable VM and extended it with a RVM API implementation on the Magali chip. In the following paragraphs we summarize our technical implementation choices.

A. Choice of a Virtual Machine

We consider that a relevant candidate virtual machine for our experiment should meet a set of criteria:

- Open source and available for a large set of classical embedded processors (or at least easy to port using a classical C compiler). For our tests an ARM11 version was needed.
- Small memory footprint and high performance (“lightweight”).
- Well documented, and possibly with an active development community, in case of needs for important modifications of the VM core.

- Easily extensible, at least through interfacing with libraries developed in native CPU bytecode.

After a comparative study of a large list of VMs we identified a short list of candidate VMs to be extended to RVM (see table I).

TABLE I
RVM CANDIDATES

	Lua	Neko	PythonVM	Squawk	Kaffe	LeJOS	TinyVM	NanoVM	Waba
small memory	x	x		x	x	x	x	x	x
performance	x	x		x		x	x	x	x
extensible	x	x	x						
lightweight	x		x	x	x	x	x	x	x
documented	x		x	x	x	x			

Squawk or other tiny Java VMs (implementing a little part of Java libraries) are good candidates to be extended into RVM: especially when current CPUs could incorporate dedicated coprocessors for the native execution of Java bytecode (e.g. Jazelle feature in ARM processors).

However, we have selected the Lua VM mainly because of its design principles: it is built to be fast, lightweight and easily extensible (could define a domain specific language). Moreover, we are experienced in development with Lua language and its internal functionalities.

B. RVM programming model

On each platform, the RVM also requires a platform specific software layer to access the hardware. Thus, RVM has a role of mapping the bytecode (functional view) on the hosting platform (system view). In this section, we show how telecom components are controlled by the VM. This control is done with four basic RVM primitives explained hereafter: RVM_ALLOCATE, RVM_CONFIGURE, RVM_CONNECT, and RVM_WAIT.

In addition to the hardware components, we defined a way to program a telecom algorithm in software, we refer to it as *software component*. software components give the ability to program new telecom components if they are not available on the platform. In our implementation, these software components will be executed by the RVM, in a specific thread, with best effort performance. Instantiating this type of software component consists in a POSIX thread creation. To realize this feature we needed operating system scheduling services. In our implementation, we have built RVM upon the ECOS highly configurable real-time operating system intended for embedded systems [14].

The RVM_ALLOCATE primitive instantiates a telecom component and returns a HANDLE required for the management of the component. After that, the RVM is able to configure the component in order to make it ready for processing (e.g., loading software code into DSP memory, IP registers setting,...).

As mentionned before, components are reconfigurable (parameterizable) in order to be used by different protocols. The RVM_CONFIGURE primitive is applied on a component HANDLE. it sends a configuration to dedicated memory slots available on the Magali components. In the case of a software component, we implemented this primitive by loading the module bytecode into the VM memory.

SDR requires fast dynamic reconfigurations of the interconnections between the platform units. Connecting two components, generally requires to configure communication controllers present in the hardware processing units or setting DSP/CPU registers to get/put data from/to a particular memory address (e.g. when using shared memory). We implemented the RVM_CONNECT primitive using communication configuration driver functionalities. In the case of HW blocks, it consists in configuring both source and destination components CCC controllers (described in section IV).

Finally, RVM is able to configure a computing unit to send an interrupt when it finishes its processing. To synchronize platform IPs configurations, RVM executes a RVM_WAIT primitive and waits until it receives a notification from the configured unit. In our case, this mechanism has been implemented using ECOS interrupt management low level primitives.

VI. EXPERIMENTS AND RESULTS

In this section we present the experiments we have realized in order $i)$ to functionally validate the RVM concept and $ii)$ to study the real time behaviour of our specific implementation on the Magali chip. We deduce from experimental results which part of a RVM implementation are the most limiting in terms of flexibility and computation efficiency. Finally, we discuss different possibilities to cope with these hard points and propose further improvements to our RVM.

A. Experimental Setup

We used two different test benches for our experiments. The first test bench, presented in section VI-A1, highlights the cost of component reconfiguration with and without RVM. The second test bench (section VI-A2) shows the performances of the system when the RVM is used to access and do computations on the stream of data. This latter case is used to evaluate the overhead of the proposed RVM abstraction and discuss its adequacy to a real SDR application.

1) First test bench: single operator application: The first test bench (figure 2) performs FFT operations on a variable number of data with three different configurations of our SDR platform:

1) in a native mode (fully optimized using Magali specific mechanisms),
2) using the previously described RVM programming model *without the* VM *interpretation layer* (i.e. simply using RVM primitives), and
3) using the full RVM concept (VM and programming model).

Fast Fourier Transform (FFT) is a common operation in signal processing. It is the central element of the TRX-OFDM

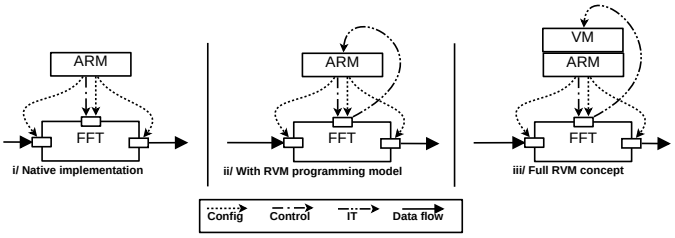

Fig. 2. Applying FFT on a variable number of data: the three configurations of test bench 1

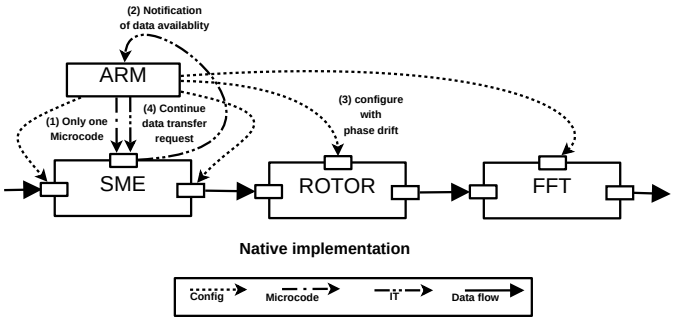

Fig. 3. CFO estimation implementation: native implementation (configuration 1)

units of the Magali chip. To realize an FFT, the Magali TRX-OFDM core must be supplied with a corresponding binary configuration. In the native programming model of Magali (configuration 1) the CCC is used to sequentially execute the same configuration in order to execute several FFTs. In this case the CPU only needs configuring the TRX-OFDM once, independently of the total number of computations. On the other hand with the proposed RVM programming model the main controller (the CPU software in configuration 2, the VM in configuration 3) has to start computation and wait for an end-of-computation IT from the TRX-OFDM at each FFT iteration (see figure 2).

2) Second test bench: IEEE802.11*a* CFO *estimation and correction:* In telecommunication systems, Carrier Frequency Offset (CFO) refers to the carrier frequency mismatch between a transmitter and a receiver, due to hardware imperfections. This well-known phenomenon is usually dealt with using CFO estimation and correction algorithms involving $sin()$, $cos()$ and $arctan()$ functions. Hardware implementations often use CORDIC IPs to efficiently handle these trigonometrical operations.

Our second test bench implements CFO estimation and correction for the IEEE802.11a protocol in pure software. In this standard, a known sequence made of a short and a long preamble is sent by the transmitter. The receiver first estimates the CFO coarsely by comparing the received short preamble to its theoretical values, then corrects data of the long preamble and proceeds with a second, more precise, estimation of the remaining CFO error.

Native implementation of this algorithm on Magali relies on a complex SME micro-code that enables synchronization with the ARM processor. The SME sends an IT when data is ready to be used by the CPU for computation. The latter "wakes up" the SME when data has been properly processed (see figure 3). Implementations with the RVM programming model use the RVM_WAIT primitive to synchronize the controller with the dataflow. A second cpu thread is responsible for the computation, either in a native ARM format (configuration 2) or using a second VM instance that executes Lua bytecode (configuration 3).

B. Results

The experiments described in the following paragraphs have been realized on the Magali chip simulation platform with a cycle-accurate ARM1176JZFS core VHDL model. CPU clock is configured to run at 362.31MHz.

1) results of test bench 1: First, we measured the total execution time for each implementation described in VI-A1. Figure 4 shows that the total execution time linearly depends on the number of processed data.

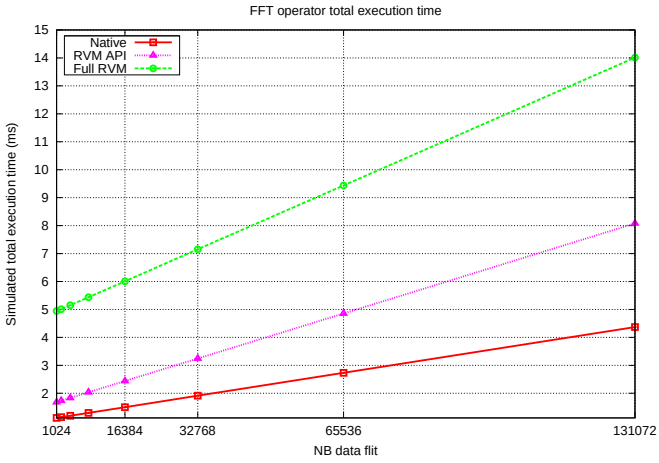

Fig. 4. Simulated total execution time

Using linear regression algorithm we can model the three curves by the linear function $y = ax+b$ where x is the number of data flit to be processed and y the simulated execution time. Couples of parameters (a, b) are listed in the following table:

TABLE II
OVERHEAD REGRESSION PARAMETERS

Implementation mode	$a(ms.data^{-1})$	$b(ms)$
Native	$2.50e^{-5}$	1.11
RVM prog model	$4.92e^{-5}$	1.63
Full RVM	$6.98e^{-5}$	4.87

Although the behaviour of the three configuration are quite similar, they correspond to very different repartitions of the time spent. To explain that we have represented in figure 5 the chronograms of work load of the CPU and of the FFT IP for each configuration

- For configuration 1 (native mode, (a) in figure 5), the complex Magali configuration sequences mentioned in

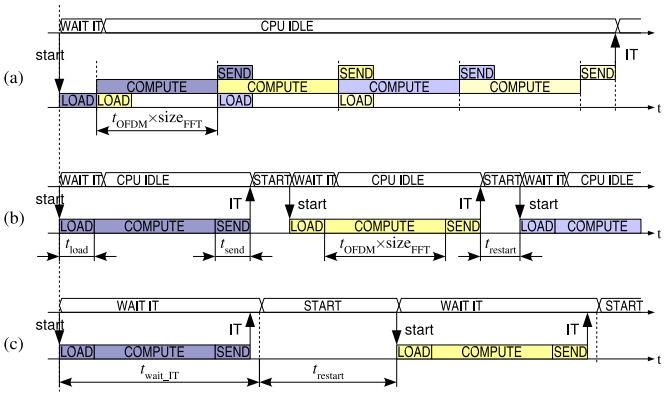

Fig. 5. Chronograms for the CPU and the FFT IP for the three configuration: native (a), RVMAPI (b), full RVM (c).

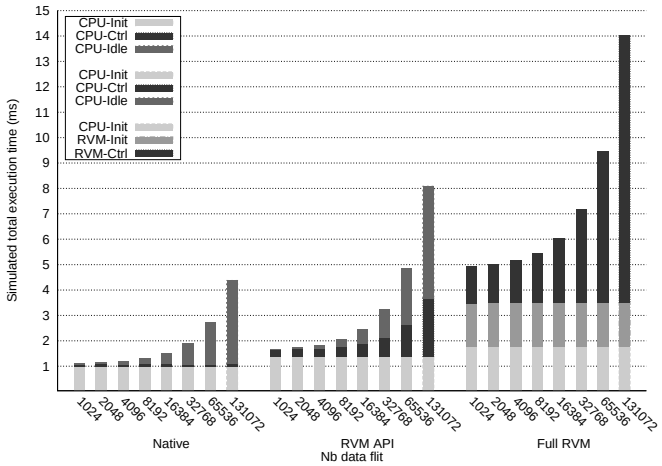

Fig. 6. Decomposition of the time spent in test bench 1 for the three configurations

section IV are used. This permits to have a single IT signal, no matter how many data are processed. This also permit to overlap communication and computation in the FFT IP. Once, initialization of the CPU is done, the time is spent in the FFT computation, hence we have:

$$a_{\text{native}} \approx a_{\text{FFT}},$$

where a_{FFT} is the time to process one data flit. (see Fig. 5(a)).

- For configuration 2 (RVM API, (b) in figure 5), the IT has to be sent at the end of each FFT computation, hence computation and communication cannot be overlapped which lead to a a total computation time of $T_{\text{rvm\_model}} = N * (t_{\text{load}} + t_{\text{FFT}} + t_{\text{send}} + t_{\text{restart}})$ for N FFT. where t_{load}, resp. t_{send}, is the necessary time for data to enter, resp. leave, the TRX-OFDM, and t_{restart} is the time needed to restart a computation when an IT is received by the CPU or, equivalently:

$$a_{\text{rvm\_api}} \approx \frac{t_{\text{load}} + t_{\text{send}} + t_{\text{restart}}}{\text{size}_{\text{FFT}}} + a_{\text{FFT}},$$

- For configuration 3 (full RVM, (c) in figure 5), the time spent in RVM primitive is more important than the time spent for executing the FFT on the IP. Hence the slope $a_{\text{full\_rvm}}$ of the line on figure 4 is only dependent of the time spent in the VM.

$$a_{\text{full\_rvm}} \approx \frac{t_{\text{wait\_IT}} + t_{\text{restart}}}{\text{size}_{\text{FFT}}},$$

where $t_{\text{wait\_IT}}$ is the time needed by the VM to switch to IT-waiting state This case clearly illustrates the cost of interpretation in the context of a VM.

Figure 6 shows the time spent by the CPU in the different program phases (initialisation, configuration of the IP, idle state, etc.) for the three configuration. This figure confirm our analysis: in the full RVM configuration, the CPU is never idle, which clearly shows that, in our implementation, the performance of the VM itself is a limiting factor for the performance of the global system.

2) IEEE802.11*a* CFO *estimation:* Figure 7 shows the total execution times for the three implementations. We notice an increasing overhead in processing time with each added layer.

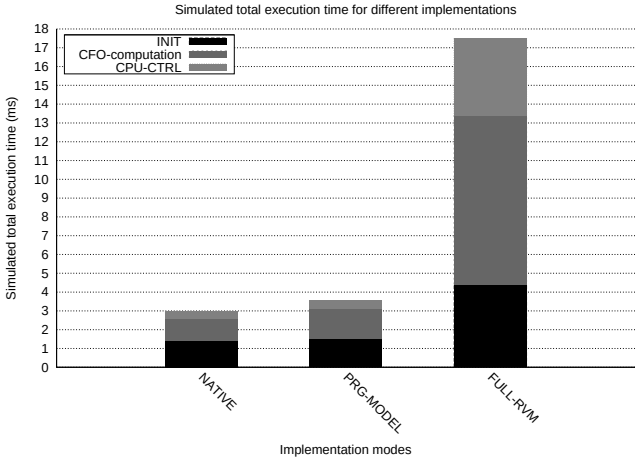

Fig. 7. CFO application execution phases

If we do not take into account the time spent in the CPU initializations (program load, boot, OS initialization, etc.) we find that RVM programming model implementation has approximatively 25% overhead compared to the native implementation, whereas full RVM implementation is approximatively 6.5 times slower than RVM programming model version.

The program size for each implementation case are detailed in the table III.

TABLE III
PROGRAMS SIZES

Implementation mode	Memory(Kb)
Native	55
RVM prog model	58
Full RVM	208

C. Discussion

It is interesting to study precisely *where* the computation time overhead comes from. We distinguish two types of overhead which results in an increase in processing time and in memory required by the SDR program to be executed. The interpretation cost for our Lua VM consists in $i)$ function calls: the name of the called function, hashed at compile time, is used to retrieve its native address, and $ii)$ transcoding native-format data from the dataflow into Lua format when the RVM has to handle computations on the dataflow.

Figure 8(a) shows that the RVM version of CFO estimation algorithm is a particularly bad case in terms of interpretation overhead, since $i)$ RVM performs repetitive calls to trigonometric functions implemented in native code, and $ii)$ data to be processed comes from the dataflow and requires to be transcoded into Lua format before and after each function call. Table IV gives a tentative decomposition of the overhead induced by the software RVM implemented here.

TABLE IV
RVM CONCEPT OVERHEAD COMPARED TO NATIVE IMPLEMENTATION `je ne comprend pas ce que veut dire Interpret. call et Interpret. comput.de plus le 200Kb ne colle pas avec les chiffres donnés plus haut`

Overhead type	memory	cpu time
Adaptation	+1 Kb	∼2 times slower
Interpret. call	+200 Kb	∼3 times slower
Interpret. comput.		∼7 times slower

This overhead could be reduced using advanced on the fly compilation techniques (Just-In-Time compilation, JIT). In these classical VM techniques the bytecode is compiled into native bytecode just when its execution is requested for the first time. It is claimed in [15] that with JIT on x86 architectures, a Lua program runs 5 times faster. We can expect roughly similar improvements on a ARM CPU. This "conventional" JIT techniques could be adapted to specific RVM context, as shown in Fig. 8(b): by using an ahead of time binary translation step. A binary translation step requires a static analysis of the code prior to its execution to generate native binary programs that can be made independent of the RVM execution loop. Translating data access code and data transcoding from native to VM format and vice-versa may be avoided and the sequence of calls may be natively realized.

On top of the arithmetical code execution time performance hit, our specific implementation of the RVM on the Magali chip does not take advantage of the hardware mechanisms provided by the CCC. These configuration registers can be used to sequence the configurations locally within a computing unit and minimize the control overhead from the RVM. The availability of these configuration registers cannot be taken for granted in a portable bytecode and are not included in the language. An ahead of time preprocessing of the bytecode using static analysis and an optimization framework including resource allocation and configuration patterns could be a solution to enhance performance. The preprocessor should

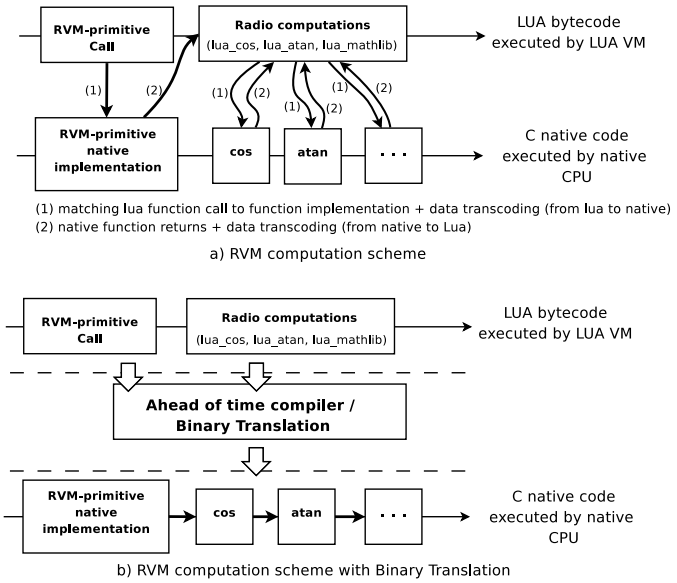

Fig. 8. Binary translation applied to a radio program

extract the sequencing of configurations to be mapped on each unit and generate CCC configurations.

Although this ahead of time binary translation or recompilation adds an extra cost for the RVM architecture, it really enables download time optimization and specialization of the portable bytecode. This step is mandatory to take advantage of specific hardware features provided by platform designers. We strongly expect from such optimisations to highly reduce the cost of the adaptation layer.

In all cases memory required by programs implementing the RVM will be higher than a native implementation. this is due to the extra code of the VM engine and the additive programming model libraries. Nevertheless, solutions like code dynamic loading (import required libraries on demand) could be experimented to reduce the amount of required memory.

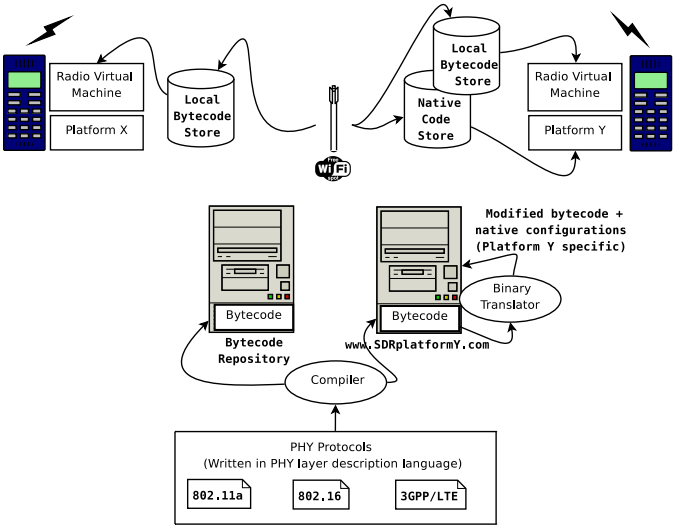

Fig. 9. RVM use case

VII. Conclusion

The different experiments introduced in this paper allows to evaluate the overheads due to the RVM concept. The first one is the necessary adaptation layer between the RVM abstract model, which intends to be generic, and the native execution model of the platform, which often targets performance. As a consequence a naive VM runtime cannot benefit from any optimized hardware mechanism specific to the model of execution of the platform. VM optimizations to take advantage of such platform accelerators may reduce this overhead, at the expanse of possibly intricated specific development (e.g., bytecode preprocessing), but that only needs to be done once.

The second limitation of the RVM compared to native development is a costly interpretation overhead, as shown in section VI-B2. This is particularly true when the VM has to proceed with computations on the flow of data. Classical VM techniques, such as "Just In Time" compilation, possibly adapted to the specific context of RVM, would greatly reduce this overhead, as would also a dedicated hardware bytecode interpretor.

As a conclusion, even if our implemented software RVM does not meet the hard real time constraints of 3/4G as it is, its adequacy to describe advanced telecommunication standards is proven and several optimizations, starting with JIT, are still to be developed to check wether sufficient performances can be achieved. Also a 3GPP-LTE waveform is currently being ported to RVM in order to set it against more complex telecommunication standard.

References

[1] I. Mitola, J., "Software radios: Survey, critical evaluation and future directions," *Aerospace and Electronic Systems Magazine, IEEE*, vol. 8, no. 4, pp. 25–36, Apr 1993.

[2] C. Grassmann, M. Sauermann, H.-M. Bluethgen, and U. Ramacher, "System level hardware abstraction for software defined radios." SDR-Forum, 2004.

[3] R. Hossain, M. Wesseling, and C. Leopold, "Application description concept with system level hardware abstraction," in *Signal Processing Systems Design and Implementation, 2005. IEEE Workshop on*, Nov. 2005, pp. 36–41.

[4] R. B. Abdallah, T. Risset, A. Fraboulet, and Y. Durand, "The radio virtual machine: A solution for sdr portability and platform reconfigurability," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–4, 2009.

[5] F. Clermidy, R. Lemaire, Y. Thonnart, X. Popon, and D. Knetas, "An open and reconfigurable platform for 4g telecommunication: concepts and application," in *12^{th} Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD'2009)*, 2009, pp. 449–456.

[6] J. Glossner, D. Iancu, M. Moudgill, G. Nacer, S. Jinturkar, S. Stanley, and M. Schulte, "The sandbridge sb3011 platform," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 16–16, 2007.

[7] A. Duller, D. Towner, G. Panesar, A. Gray, and W. Robbins, "picoarray technology: the tool's story," in *Design, Automation and Test in Europe, 2005. Proceedings*, March 2005, pp. 106–111 Vol. 3.

[8] K. van Berkel, F. Heinle, P. P. E. Meuwissen, K. Moerman, and M. Weiss, "Vector processing as an enabler for software-defined radio in handheld devices," *EURASIP J. Appl. Signal Process.*, vol. 2005, pp. 2613–2625, 2005.

[9] D. Tucker and G. Tagliarini, "Prototyping with gnu radio and the usrp - where to begin," in *Southeastcon, 2009. SOUTHEASTCON '09. IEEE*, March 2009, pp. 50–54.

[10] G. Minden, J. Evans, L. Searl, D. DePardo, V. Petty, R. Rajbanshi, T. Newman, Q. Chen, F. Weidling, J. Guffey, D. Datla, B. Barker, M. Peck, B. Cordill, A. Wyglinski, and A. Agah, "Kuar: A flexible software-defined radio development platform," in *New Frontiers in Dynamic Spectrum Access Networks, 2007. DySPAN 2007. 2nd IEEE International Symposium on*, April 2007, pp. 428–439.

[11] D. Lattard, E. Beigne, F. Clermidy, Y. Durand, R. Lemaire, P. Vivet, and F. Berens, "A reconfigurable baseband platform based on an asynchornous network-on-chip," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 1, pp. 223–235, 2008.

[12] J. Martin, C. Bernard, F. Clermidy, and Y. Durand, "A microprogrammable memory controller for high-performance dataflow application," in *35^{th} European Solid-State Circuit Conference (ESSIRC'2009)*, 2009, pp. 348–351.

[13] F. Clermidy, Y. Thonnart, R. Lemaire, and P. Vivet, "A communication and configuration controller for noc based reconfigurable data flow architecture," in *3^{rd} IEEE International Symposium on Networks-on-Chip (NoCs'2009)*, May 2009, pp. 153–162.

[14] A. Massa, *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002.

[15] W. C. Luiz Henrique de Figueiredo and R. Ierusalimschy, *Lua Programming Gems*. Lua.org, December 2008.

[16] Y. Durand, C. Bernard, and D. Lattard, "FAUST: On-chip distributed architecture for a 4g baseband modem SoC," in *Design & Reuse IP-SoC, Grenoble, France*. Grenoble, France: IEEE Computer Society, Dec. 2005.