# Development Framework for Implementing FPGA-Based Cognitive Network Nodes

Jörg Lotze*, Suhaib A. Fahmy*, Juanjo Noguera†, Barış Özgül*, Linda Doyle* and Robert Esser†
*CTVR, Trinity College Dublin, Ireland
Email: {jlotze, suhaib.fahmy, ozgulb, linda.doyle}@tcd.ie
†Xilinx Research Labs
Email: {juanjo.noguera, robert.esser}@xilinx.com

***Abstract*—This paper identifies important features a cognitive radio framework should provide, namely a virtual architecture for hardware abstraction, an adaptive run-time system for managing cognition, and high level design tools for cognitive radio development. We evaluate a range of existing frameworks with respect to these, and propose a novel FPGA-based framework that provides all these features. By abstracting away the details of hardware reconfiguration, radio designers can implement FPGA-based cognitive nodes much as they would do for a software implementation.**

**We apply the proposed framework to the design and implementation of a receiver node that works in two modes: discovery, where it uses spectrum sensing to find a radio transmission, and communication, in which it receives and demodulates the said transmission. We show how the whole design process does not require any hardware experience on behalf of the radio designer.**

## I. Introduction

Autonomous, self-organising networks are built upon smart nodes that can establish communication with peers without the presence of a fixed network control channel [1]. For nodes to be able to join such networks, they must be able to not just communicate using the necessary protocols, but to assess existing communication activity within the network. To support these capabilities, a communications platform[1] must provide computational resources sufficient to handle signal detection and analysis algorithms in addition to complex modern communications protocols. We are concerned here with detection and analysis algorithms involved in setting up networks as exhibited in ad-hoc network and femto cell implementations, rather than those defined in protocols. These two phases – discovery and communication – are typically distinct, with discovery only required on introduction to, or change in the network.

Emerging software radio frameworks[2] enable cognitive nodes to be reconfigurable at runtime. However, most work has focused on software running on PCs and many of the techniques used do not scale down to embedded systems well, due to high computational complexity. Signal classification and feature extraction are even more of a challenge on embedded platforms. Custom embedded hardware, typically using field-programmable gate arrays (FPGAs), provide an oft-lauded platform for high performance embedded systems. While there have been many examples of FPGA-based wireless systems, the design approach has been largely ad-hoc.

FPGAs allow for the design of custom computational architectures that can exploit parallelism in many algorithms, resulting in a significant speedup over software implementations. One of the capabilities of FPGAs that is of particular use in the area of cognitive nodes is partial reconfiguration. This allows FPGA functionality to be changed at runtime, allowing parts of the application to be replaced while other parts continue to function. Using this capability is a complex manual task that involves engineers who understand low-level FPGA technology. However, it is by exploiting this capability that high performance cognitive nodes with many configuration options can be realised efficiently on a small device.

We believe that a practical framework for embedded cognitive nodes must address three aspects:

- A *virtual architecture*, which can be mapped to different boards and FPGA devices. Whatever is built on top of this virtual architecture is independent of the underlying FPGA device. This allows us to select the physical platform that is most suited to the application in question.
- An *adaptive runtime system* that manages cognition and reconfiguration at run-time to support custom cognitive control while hiding low-level reconfiguration details.
- High level *design tools* for implementing cognitive nodes. The key consideration is that radio designers with no FPGA or hardware design experience should be able to design a radio that takes advantage of the underlying FPGA features (e.g., performance).

The framework we present here addresses these three aspects, greatly simplifying the design and implementation of FPGA-based cognitive nodes. To the best of our knowledge, no framework found in the literature addresses all these aspects.

An analysis and evaluation of a selection of existing frameworks is given in Section II. Our proposed framework is presented in Section III. Section IV applies this framework to the design of a cognitive network node switching between spectrum sensing and reception on the same hardware. Section V draws conclusions.

[1] "platform" is used to denote physical hardware throughout this paper.

[2] "framework" is used to denote the architecture and tools for design and execution of software radios or cognitive radios.

978-1-4244-4148-8/09/$25.00 ©2009
This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE "GLOBECOM" 2009 proceedings.

## II. Existing Software Radio Frameworks

In this section we give a brief overview of a selection of software radio frameworks that closely relate to the work presented in this paper. They are assessed in the discussion, based on the requirements raised in Section I.

### A. Existing Frameworks

*1) GNU Radio:* GNU Radio is an open source software development toolkit for software radio applications [2]. It runs on general purpose processors, allowing for easy, low-cost software radio experimentation and development. Radio applications are written in Python, while the performance-critical signal processing components, provided, are compiled from C++. A large number of components are available in GNU Radio, but it is a pure software platform, not designed for low-power, high performance embedded systems.

*2) Open-Source SCA Implementation::Embedded:* OSSIE is a lightweight Software Communication Architecture (SCA)[3] implementation [3]. The SCA is an interoperable, multi-platform architecture framework for software radio systems. OSSIE is implemented purely in C++ for execution on a general purpose processor, allowing fast prototyping for experimentation and research. A library of pre-built components and waveforms is available. An extension of OSSIE for embedded systems is planned for 2009.

*3) IRIS:* IRIS [4] is a flexible and reconfigurable framework for Windows on x86 processors[4]. Radio flow graphs for IRIS are described in an XML file. A Decision Engine can be implemented by the radio designer to subscribe and react to events triggered by the radio components. Reactions can be everything between diagnostic output and a full reconfiguration of the radio flow graph. The IRIS engine, the component library and the Decision Engine are compiled from C++. As GNU Radio and OSSIE, IRIS is not designed as a framework for embedded cognitive radio systems.

*4) Wireless Open-Access Research Platform:* WARP is a scalable, extensible and programmable hardware platform with a Xilinx Virtex-II Pro FPGA as its baseband processor and up to four RF daughter boards [5]. The physical layer of a radio is implemented in the FPGA logic fabric, while MAC layer functionality can be implemented in C using the embedded PowerPC processor cores without an operating system. It allows efficient software radio implementations using low-level FPGA design tools. However, it does not provide an adaptive run-time system for cognition or high-level FPGA design tools.

*5) Lyrtech:* Lyrtech offers a variety of SDR development platforms, together with software and hardware development kits [6]. The radio architecture used is based on the SCA, extended to support FPGA and DSP components [7]. Lyrtech's hardware platforms are modular, consisting of antennas, an RF module, an interface module, a baseband processor and optional expansion modules. The baseband processor is a hybrid system with a DSP processor, an ARM processor, and a Xilinx FPGA. A software development kit for DSP and ARM development is provided, along with an extension of Xilinx System Generator for model-based FPGA design. The FPGA, DSP, and ARM are capable of run-time reconfiguration, but no design or run-time framework for cognition is provided.

*6) Kansas University Agile Radio:* KUAR is a compact, powerful, and flexible software radio development and experimentation platform for cognitive radio research [8]. It consists of a full embedded Pentium PC running the Linux operating system, a Xilinx Virtex-II Pro FPGA, an RF front-end, and active antennas. Its small form factor and the optional battery pack make it easily portable. KUAR provides software and hardware APIs for configuration and control and for processor-FPGA communication. Radio applications are developed using standard compilers and low-level FPGA design tools. A small library of radio components is provided. Radio design is handled at a low level, i.e., no high level toolkit for constructing radio flow graphs or handling reconfiguration is provided.

*7) WINLAB Network Centric Cognitive Radio:* The WiNC2R is a cognitive radio platform [9] that uses flexible hardware accelerators to achieve programmability and high performance at each layer of the protocol stack. The prototype consists of one or more baseband modules, each connected to an RF module, a networking module, and a CPU. The performance intense radio functions are executed on dedicated hardware accelerators (implemented in FPGA logic), while control intensive functions are performed by data processors (implemented as soft CPU cores on the FPGAs). A system scheduler manages the synchronisation of all the processing elements as well as the data transfers. System reconfiguration is managed by the external CPU which can reconfigure hardware accelerators, data processors, and the system scheduler if required. The WiNC2R is still in an early stage, the prototype is not fully developed and no design tools are available so far.

### B. Discussion

As mentioned in Section I, cognitive radio frameworks should provide a *virtual architecture* for hardware abstraction and an *adaptive run-time* and *design framework* for management and design of cognition. For realistic cognitive radio applications, the platform should be embedded and low-power. An evaluation of each of the discussed platforms with respect to these criteria is given in Table I.

Pure software frameworks like GNU Radio, OSSIE and IRIS have excellent design frameworks and are easy to use, making them popular. However, processing performance, power consumption and large form factor limit the application areas severely, in contrast to embedded hardware frameworks.

Hardware frameworks show various levels of reconfigurability, but lack a structured and easy-to-use design-flow for radio design. Runtime reconfigurability is a requirement for cognitive radios but the platforms that support it are cumbersome to develop for. While Xilinx FPGAs support partial reconfiguration, doing so remains complex. Clearly, a

[3]the SCA is at the core of the JTRS project, hosted by the U.S. DoD

[4]based on the IRIS 2007 version; for the current version, see Section III.

978-1-4244-4148-8/09/$25.00 ©2009
This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE "GLOBECOM" 2009 proceedings.

TABLE I
SUMMARY OF THE DISCUSSED PLATFORMS.

| | Virtual Arch. | Adaptive Runtime | Design Tools | Recon-figurable | Low-power Embedded |
|---|---|---|---|---|---|
| GNURadio [2] | ○[1] | ● | ● | ● | × |
| OSSIE [3] | ● | × | ○[2] | ○ | ○ |
| IRIS [4] | ○[1] | ● | ● | ● | × |
| WARP [5] | ×[3] | × | × | ● | ● |
| Lyrtech [6] | ○[3] | × | ○ | ● | ● |
| KUAR [8] | ● | ○ | ○ | ● | ● |
| WiNC2R [9] | ○[4] | ● | × | ● | ● |

● : fully supported; ○ : partly supported; × : not supported

[1] no specific hardware system architecture (operating system)
[2] the design framework does not include cognition
[3] is a pure hardware platform, no portable architecture
[4] uses CPU & FPGA (but no definition of arch. & no prototype yet)

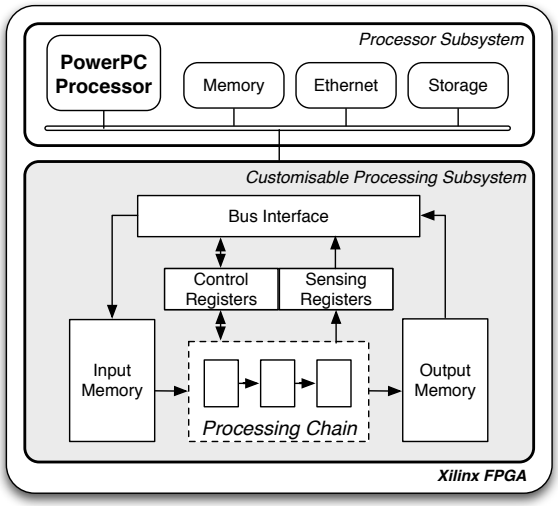


Fig. 1. The virtual architecture. The Processor Subsystem contains basic hardware for running Linux, the Customisable Processing Subsystem is used for radio configurations, which can be reconfigured at run-time.

combination of the flexibility of software frameworks with the performance of FPGA-based systems is required.

## III. Proposed Cognitive Radio Framework

It is clear from the discussion in Section II that developing dynamically reconfigurable cognitive nodes using FPGAs remains a specialist area, requiring detailed knowledge of FPGA architectures, tools and hardware design in general. We address this challenge in three parts, as outlined in Section I, that together form our proposed framework; a virtual architecture, an adaptive run-time, and high-level design tools for implementing cognitive radios.

### A. Virtual Architecture

The virtual architecture generalises our approach and consists of two parts: the *Processor Subsystem (PS)* and the *Customisable Processing Subsystem (CPS)*, as shown in Fig. 1.

The *PS* integrates the hardware modules required to run a standard Linux operating system. In addition to the embedded PowerPC processor and Ethernet modules, the PS includes a memory controller to interface to external DRAM, and an interface to an external storage device for the file system (e.g., CompactFlash).

The PS is responsible for two main tasks: executing the software run-time system in charge of the cognition process (see Section III-B); and executing computationally non-intensive components in the radio chains.

The *CPS* is used for implementing radio components with high computational requirements in hardware (i.e., FPGA logic). It is implemented in the FPGA fabric as a partially reconfigurable region, which can be re-configured at run-time without disrupting the execution of the processor subsystem. The reconfiguration is managed through a Linux device driver. The customisable processing subsystem is attached to the on-chip bus and is mapped into the address space of the PowerPC (see Fig. 1). The number of CPSs is not limited to one; for the sake of simplicity we assume a single region throughout this paper.

In order to transfer data into and out of the baseband processing subsystem, the first and last components in a chain are connected to on-chip memories. A set of interface registers, accessible from software, is used for controlling hardware execution, accessing component parameters and sensing events. When the radio front-end can be connected directly to hardware, the input or output memory can be replaced by the front-end controller, for receive or transmit operation, respectively. The processing chain is synthesised using the high-level design tools described in Section III-C.

This virtual architecture makes our framework generic and independent of the final target board or specific FPGA used, and hence portable. The only prerequisite is to have a hardware platform which can execute Linux.

### B. Adaptive Runtime System

The principle idea in this software architecture is to separate the processing and control planes. The processing plane consists of baseband processing components that perform the computation required for the radio system. These can then be mapped to either the CPS in the virtual architecture, or to the processor. The control plane is where the reasoning, adaptation and self-organising occurs; it manages the radio. Fig. 2 shows the architecture of the Runtime System.

The flow graph of the processing plane, containing the processing components, which are selected from a pre-existent library, and the flow of data between them, is described in an XML file. To allow for re-use in different applications, each component has a set of parameters and a simple interface to the user-created *Decision Engine*.

IRIS, as described in Section II-A3, has been extended to run on embedded Linux and to support hardware components (i.e., components implemented in the FPGA logic fabric). We have adopted a Linux on FPGA implementation for embedded PowerPCs, as offered by Xilinx [10].

978-1-4244-4148-8/09/$25.00 ©2009
This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE "GLOBECOM" 2009 proceedings.

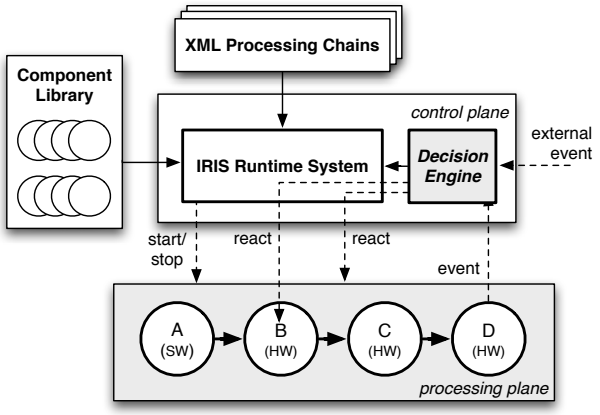


Fig. 2. The Runtime Software Architecture.

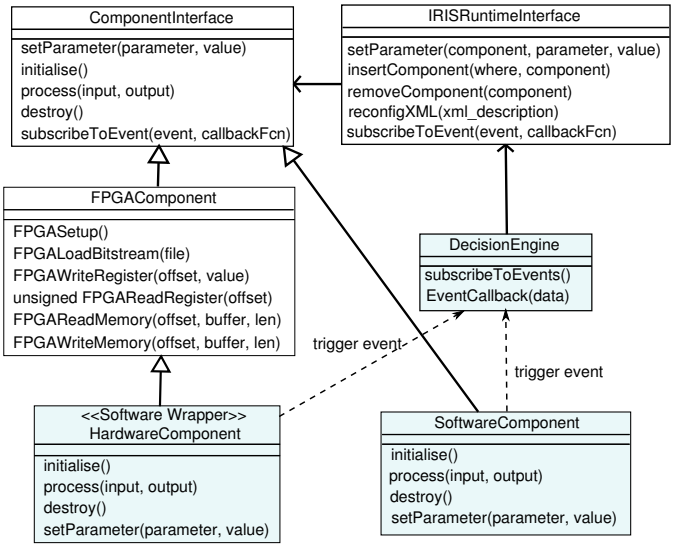


Fig. 3. Class diagram for software components and automatically generated wrappers for hardware chains, along with the Decision Engine interface.

Since Xilinx FPGAs allow for runtime partial reconfiguration, it is possible to change the configuration of only the CPS, and hence the function of the system at runtime from within the IRIS application.

System execution is managed by the *IRIS Runtime System*, which parses the specified XML file, instantiates and connects the required components, loads the Decision Engine, and controls and monitors system execution. Thus, both the Decision Engine and the Runtime System together form the control plane, as shown in Figure 2.

Based on this software architecture, the only two inputs that the user needs to specify are: the XML radio chains; and the C++ Decision Engine. The Decision Engine does not need to consider implementation details of the components it is controlling (i.e., software or hardware implementation), since the interfaces are abstracted as per the class diagram for software components and hardware wrapper components shown in Fig. 3 and it is only accessing the running radio through the *IRISRuntimeInterface*. This accesses components through the *ComponentInterface*, which all processing components must inherit from. Since the Decision Engine is a software component, cognitive algorithms are limited by the processing power and capabilities of the PowerPC.

Processing components can be described in software using C++, or in hardware using a hardware description language. We do not deal with the high-level design of components, as this can be done using existing tools. To allow a common interface to both types of components from the control plane, one or more hardware components are wrapped in a *software wrapper* with an identical interface to software components. The class *FPGAComponent* provides a set of FPGA operations for software wrappers, such as methods for accessing registers and embedded memories. All hardware components implement a common interface based on a simple FIFO to allow direct connection of consecutive components to chains.

Our aim is to maintain identical behaviour and mechanisms for processing chains that are implemented in the FPGA logic fabric as for those in software. This requires the Decision Engine to be able to sense events from individual hardware components, and to be able to apply changes to individual components and to the chain as a whole, using the same methods as for a software chain.

Events can be triggered from the hardware in the partial region by setting a value in a sensing register. The wrapper checks these registers regularly and triggers a software event to the Decision Engine which can then react. Reactions include *parametric*, *structural* or *functional* reconfigurations, as defined below, and are effected through the Runtime System via the *IRISRuntimeInterface*.

It is important to distinguish between application level adaptation and its manifestation in terms of hardware reconfiguration. Clearly, *functional reconfiguration* entails changing the functionality of the system and hence, replacing the whole processing chain. This simply maps to a hardware reconfiguration of the whole region, which is achieved by calling the *FPGALoadBitstream* method in the implementation of the *initialise* method.

*Structural reconfiguration* involves the replacement of components or the introduction of new components. In an ideal scenario, this could map to a partial reconfiguration of just the necessary part of the chain. If one CPS holds multiple components, represented by one wrapper from the software perspective, the whole region must be reconfigured (a call to *FPGALoadBitstream*). While it is possible to implement a hardware chain that contains multiple alternatives of a component and uses a multiplexer to select between them, this clearly wastes resources, while precluding the need for, and foregoing the benefits of, partial reconfiguration.

*Parametric reconfiguration* is the change of a parameter of one of the components within the processing chain and is more complex. The *setParameter* method in the software wrapper of a CPS is used. Parameters can for instance include the gain of a scaling component or the generating polynomial of a convolutional code. In some cases, where changing the

978-1-4244-4148-8/09/$25.00 ©2009
This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE "GLOBECOM" 2009 proceedings.

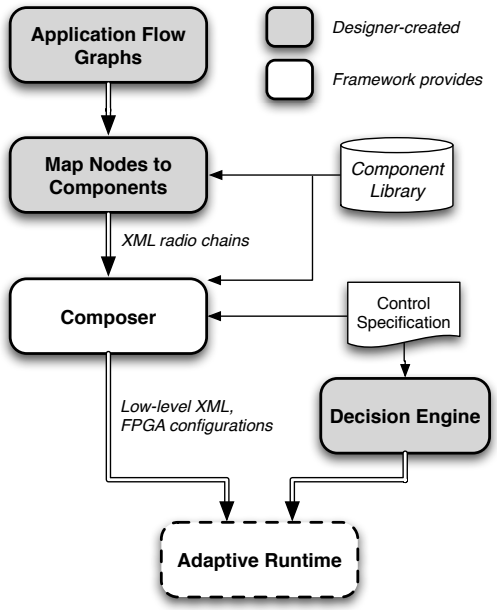


Fig. 4. Proposed Design Flow.

parameter has minimal impact on the computational circuitry, setting a parameter can simply map to a write into a software accessible register (a call to *FPGAWriteRegister*). An example is a scaling component where the gain is reconfigurable. By introducing a register that holds the gain to be used by the hardware, a reconfiguration of this parameter can be made without the need for an FPGA reconfiguration. However, if the parameter to be changed implies considerable changes in the hardware circuitry, e.g. generating polynomial of a Viterbi decoder, it maps to an FPGA reconfiguration (a call to *FPGALoadBitstream*).

When the run-time system initialises the software wrapper, the input and output memories as well as the register addresses of the Virtual Architecture are mapped into the Linux virtual memory space (*FPGASetup* operation in Figure 3). A bitstream is loaded into the CPS of the FPGA using the Linux driver (*FPGALoadBitstream*). When the execution reaches the software wrapper, data is transferred to the hardware input memory, then the hardware is enabled using the register interface. Once execution has completed, data can be read back from the output memory by the software wrapper. The software wrapper is written such that it allows for processing of variable sized blocks of data (i.e., multiple iterations are completed if necessary).

### C. High-Level Design Tool Flow

The tools provided in this framework facilitate system-level design and composition of radios using components existent in the component library. To design a cognitive node for the proposed platform, radio designers follow the flow given in Fig. 4.

The two aspects of a cognitive radio design are the processing chains and the cognitive process. The processing chains are typically described in a flow graph of radio components, representing the first step in every design. The designer maps the desired components to those available in the component library, and describes the radio using an XML file. At present, a large number of components are available as software, with a growing number of hardware components being developed.

A *Control Specification* is used to specify which component parameters are required to be runtime reconfigurable and which values they can take. This means parameters which are not going to be reconfigured can be fixed in the generated FPGA configurations, and no logic for interfacing them to software is needed. Thus, since we map each chain to a single reconfigurable region, the total number of possible configurations required is the product of the number of possible configurations for all components. Hence it is important to use the control specification to limit the values of the parameters and hence the number of FPGA configurations generated.

The *Composer* tool uses the Control Specification, the XML radio chains and the component library to generate the FPGA configurations. First, it determines which components in the radio chains to implement in hardware, based on the availability of hardware implementations and the available hardware resources. It then merges consecutive hardware components and wraps them in a single software component, represented in a low-level XML representation of the radio chain. The software wrapper represents the software interface to the hardware, handling the data transfers to and from hardware, and abstracting and managing the FPGA reconfiguration through the *FPGAComponent* base class.

The *Composer* then generates the required FPGA bitstreams, which along with the low-level XML files and wrapper components are used by the Runtime System.

The designer uses the Control Specification to write the Decision Engine, in C++, which can subscribe to events raised by components, or externally, and apply functional, structural or parametric reconfiguration to the processing chain. The Decision Engine uses the parameters and events of the components as represented by the original XML radio chains.

It is possible to add custom components to the component library by following existing hardware design flows. Additionally, there exist numerous commercial tools that allow high-level design of signal processing modules (e.g., AccelDSP synthesis tool is a high-level MATLAB-based tool for designing DSP blocks for Xilinx FPGAs).

Since IRIS supports both software and hardware components, we can combine the two within a single chain, and this provides a powerful verification and debugging mechanism. It is possible to process data through hardware, then store it in a file for analysis, or transfer the data for visualisation in MATLAB. This is also beneficial when implementing a custom hardware component, since data can be compared to software implementations.

In the following section, we describe the design and implementation of a real-world cognitive radio application using our proposed framework.

978-1-4244-4148-8/09/$25.00 ©2009
This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE "GLOBECOM" 2009 proceedings.

## IV. Real-World Application

### A. Sensing for Cognitive Networks

Spectrum sensing is essential for cognitive networks that share frequencies with other (legacy) networks to avoid interference. It shows great potential for increasing spectrum utilisation. A node can use spectrum sensing to detect bands of spectrum that are being occupied or vacant, and use this to decide where to transmit without causing interference. In this section, we use the framework proposed in Section III to implement a basic network link, where a receiver node uses sensing to locate a transmission at unknown frequency. The application presented here is being developed as part of an implementation for LTE femtocells for GSM frequency refarming.

### B. Design of a Cognitive Sensing Node

In this application, a transmitter, also implemented with the proposed framework, is set up to transmit at an unknown arbitrary frequency. The receiver node wakes up, and enters sensing mode, where it uses energy detection based spectrum sensing to locate transmitters within the frequency band. Once a transmission has been found, the radio reconfigures into reception mode, and attempts to demodulate the transmission. If this fails, the receiver switches back into sensing mode, and searches for another signal. Once a correct transmission is found, the receiver continues to demodulate. The application data is provided by the VideoLAN VLC player via video streaming over UDP.

The basic application flow graph consists of two chains, one containing a power spectral density (PSD) estimator component followed by a detection component that finds the centre frequency of any transmissions detected, and another containing a full receiver chain. Within the receive chain, we use the output of a deframer to check for frame access codes inserted at the transmitter. If these are found for each data frame, the reception frequency is correct. If not, a 'no signal' event is raised, which is used by the Decision Engine to switch to sensing mode, and the other chain. Figure 5 shows the data and control flow of the application to be designed.

The receive chain components were all pre-existent in the component library as the result of previous demos we have designed. However, we had to design a PSD estimation component in hardware from scratch. We used VHDL and System Generator to enable MATLAB-based simulation.

The basic outline of the PSD estimator is as follows: The complex baseband signal is passed through an FFT, with adjustable point size. The square magnitude of the FFT output is calculated, resulting in the instantaneous PSD for that FFT window. To obtain a a less noisy PSD estimate, we average over an adjustable number of windows. We use the Xilinx FFT Core and a Xilinx memory core.

To estimate the centre frequency of possible transmitters in the detector component, the PSD is correlated with the spectral footprint of the expected transmission signal. We use the thresholding method described in [11] to find peaks in the correlated PSD, which are reported to the Decision Engine. The detector is implemented in software, as it is not processing intensive and only runs once after PSD estimation.

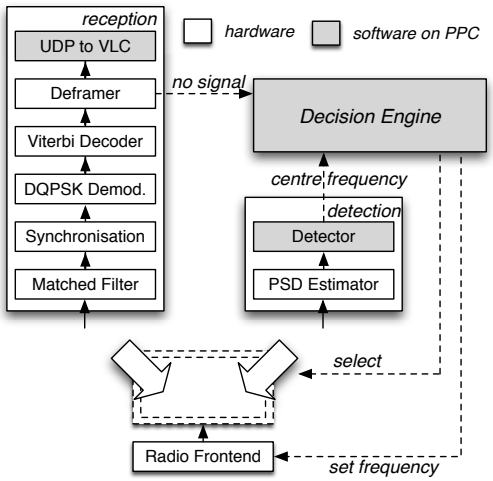


Fig. 5. Reception and detection chains with Decision Engine and control signals. The Decision Engine selects which of the two chains are loaded into the system and connected to the radio frontend.

Now all the required components are available, we write two XML chains, one for each of the sensing and receiving chains. The Control Specification lists the events 'no signal' and 'centre frequency', the two XML chains, and the 'frequency' parameter of the radio front-end. We then write the Decision Engine, in C++, which subscribes to the 'no signal' event generated by the Deframer and the 'centre frequency' event generated by the Detector (Fig. 5). The system starts in detection mode, scanning for possible signals. The Decision Engine uses the centre frequencies reported by the Detector to tune the radio front-end's frequency and to switch to reception mode. If no signal is found, the Deframer will trigger the 'no signal' event, which causes the Decision Engine to tune to the next frequency from a previous detection run, or to switch to detection mode again and scan for new frequencies, possibly in another frequency band.

This represents all user input required to implement this system. The designer has not had to think about hardware implementation details, has not had to partition the hardware into separate configurations, nor map the reconfigurations to FPGA bitstreams.

Now the Composer takes the chains and the Control Specification, decides for hardware/software partitioning, as illustrated in Fig. 5, and builds the FPGA processing chains from the components in the library. It exposes the parameters of and events specified in the Control Specification and combines the hardware portion of each chain into a software wrapper component. Low-level XML descriptions of both radio chains are generated, explicitly containing the software wrappers. The Composer then connects the required signals for the Decision Engine to function, and generates the FPGA configurations.

Now the radio can be executed and the system will correctly

978-1-4244-4148-8/09/$25.00 ©2009
This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE "GLOBECOM" 2009 proceedings.

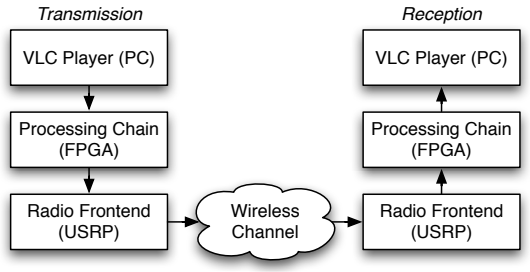


Fig. 6. Sensing Application Setup.

apply the different types of runtime reconfiguration as discussed in Section III-B. This is without the designer worrying about low-level detail, and yet, the system takes advantage of the computational power of the FPGA, and of dynamic partial reconfiguration.

Reconfiguration of the FPGA takes a time in the order of tens of milliseconds, which is easily sufficient for applications of this nature, where reconfiguration is expected to occur over much longer time periods.

This reference implementation has been synthesised for the Xilinx XUP development board, with a Virtex II-pro FPGA [12]. Due to the virtual architecture, we could use any other development board with sufficient resources to run Linux, e.g. the WARP board discussed in Section II-A4. In this example, we use the USRP as radio frontend [13], which is accessed through a software component. The resulting system setup is shown in Fig. 6.

### C. Discussion

The radio designed in this section has been successfully demonstrated using a live video stream. The receiver is able to find the transmission frequency without prior information, at startup or if the transmitter changes frequency during transmission. The PSD estimator and detector components can easily be modified to detect unused spectrum for use by the transmitter. In the scenario of an LTE femtocell deployed in GSM frequency bands, interference to neighbour femtocells can be significantly reduced by sensing for their transmission bands using a modified version of the sensing mode used here. This demonstrator can also be extended to include advanced signal classification algorithms to inform the Decision Engine how to configure the receive chain, e.g., setting the modulation type or data rate. The different settings might result in different FPGA configurations loaded into the CPS but the Decision Engine is not concerned with this level of detail.

## V. Conclusions and Future Work

The major requirements for a cognitive radios development framework have been identified and implemented in the proposed framework, targeting Xilinx FPGAs. The novel framework consists of three key building blocks: a virtual architecture for hardware abstraction, an adaptive run-time system for managing cognition, and high-level design tools for cognitive radio development.

The computational performance and run-time partial reconfigurability of FPGAs are harnessed, while the low-level implementation details are hidden from the designer. That is, the designer is focused in the development of the cognitive radio functionality (i.e., radio chains and cognitive engine), and not in the low-level FPGA implementation details.

The cognitive engine is implemented in software (i.e., embedded PowerPC), without being aware of where the specific components of the processing chain are implemented (i.e., embedded PowerPC or FPGA logic fabric). Using an example cognitive receiver implementation that switches between a sensing and a receiving mode on the same FPGA device, we have demonstrated the framework's capabilities.

Our main aim at present is to continue work on the *Composer*, to improve its level of automation. We are also exploring new methods and high-level tools for the design of hardware components for incorporation into this framework.

## Acknowledgments

The authors acknowledge Stephen Neuendorffer and Kees Vissers of Xilinx Labs for their support in providing the Linux on FPGA implementation used in this work.

This research project IP20060367, is funded by Enterprise Ireland under its Innovation Partnership scheme.

## References

[1] P. D. Sutton, K. E. Nolan, and L. E. Doyle, "Cyclostationary signatures in practical cognitive radio applications," *IEEE J. Sel. Areas Commun.*, vol. 26, no. 1, pp. 13–24, Jan. 2008.

[2] Free Software Foundation, Inc. (2008) GNU radio - the GNU software radio. [Online]. Available: http://www.gnu.org/software/gnuradio/

[3] M. Robert *et al.*, "OSSIE: Open source SCA for researchers," in *SDR Forum Technical Conference and Product Exposition (SDR Forum)*, Phoenix, Arizona, USA, Nov. 2004.

[4] P. MacKenzie, "Software and reconfigurability for software radio systems," Ph.D. dissertation, Trinity College Dublin, Ireland, 2004.

[5] K. Amiri *et al.*, "WARP, a unified wireless network testbed for education and research," in *IEEE International Conference on Microelectronic Systems Education (MSE)*, San Diego, CA, USA, 3–4 Jun. 2007.

[6] R. Sathappan, M. Dumas, L. Belanger, and M. Uhm, "New architecture for development platform targeted to portable applications," in *SDR Forum Technical Conference (SDR)*, Orlando, Florida, USA, 2006.

[7] J. Jacob and M. Dumas, "CORBA for FPGA the missing link for SCA radios," Lyrtech, Quebec, Canada, white paper, Feb. 2007.

[8] G. J. Minden *et al.*, "KUAR: A flexibile software-defined radio development platform," in *Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, Dublin, Ireland, 17-22 Apr. 2007.

[9] Z. Miljanic, I. Seskar, K. Le, and D. Raychaudhuri, "The WINLAB network centric cognitive radio hardware platform – WiNC2R," *Mobile Networks and Applications*, vol. 13, no. 5, pp. 533–541, Oct. 2008.

[10] S. Neuendorffer and C. Epifanio, "Generic partially reconfigured processor systems applied to software defined radio," in *SDR 07 Technical Conference and Product Exposition*, 2007.

[11] T. J. O'Shea, T. C. Clancy, and H. J. Ebeid, "Practical signal detection and classification in GNU radio," in *SDR Forum Technical Conference (SDR)*, Denver, Colorado, USA, Nov. 2007.

[12] *Xilinx University Program Virtex-II Pro Development System – Hardware Reference Manual*, Xilinx, Inc., Mar. 2005. [Online]. Available: http://www.xilinx.com/univ/XUPV2P/Documentation/XUPV2P_User_Guide.pdf

[13] *Universal Software Radio Peripheral – The Foundation for Complete Software Radio Systems*, Ettus Research LLC, Mountain View, California, USA, Nov. 2006. [Online]. Available: http://www.ettus.com/downloads/usrp_v4.pdf

978-1-4244-4148-8/09/$25.00 ©2009
This full text paper was peer reviewed at the direction of IEEE Communications Society subject matter experts for publication in the IEEE "GLOBECOM" 2009 proceedings.