An Architecture for Software-defined Radio Implementation on Embedded Systems

Tiago Rogério Mück   Roberto de Matos   Antônio Augusto Fröhlich

Federal University of Santa Catarina (UFSC)
Laboratory for Software and Hardware Integration (LISHA)
{tiago, roberto, guto}@lisha.ufsc.br

Abstract

There are a lot of wireless communication protocols being used, and this results in a series of difficulties for developers of embedded systems that interacts with devices which can use different communication protocols. Software-defined radios (SDR) aim to solve this problem by using a software-based approach to provide flexibility for the implementation of the protocol's physical layer. In this paper, an analysis of SDR implementation approaches are presented, and an architecture for SDR implementation on embedded systems is proposed. The architecture uses hybrid HW/SW components and programmable logic devices in order to enable the efficient implementation of software-defined radios from high-level models. The results show that the architecture imposes a very low and deterministic overhead on the SDR processing chain.

1. Introduction

There are a lot of wireless communication protocols being used, and this yields several difficulties for developers of embedded systems that interact with devices which can use different communication protocols. Providing adaptability for the communication protocols is a goal to be achieved. However, the hardware-based architecture of traditional radios imposes several limitations on providing this adaptability.

In traditional radios, each element of the receive/transmit hardware chain has a specific function, and the components are designed to work according to a fixed protocol. When a protocol's parameter change, the information is not transmitted correctly by the radio anymore. Hardware components need to be replaced so the system can work with new standards. This lack of flexibility leads to higher development costs and time-to-market.

Software-defined radios (SDR) follow a software-based approach to eliminate the limitations imposed by the traditional radios. The idea behind SDRs is to use a multi-band system which allows for transmitting and receiving in the frequencies of interest, called RF front-end, and to do all the modulation and demodulation of the signal in software instead of hardware, thus allowing for more flexibility in implementing the physical layer of the communication protocols. Figure 1 shows the general architecture of an SDR. Analog-to-digital/Digital-to-analog converters are used to make the interface between the analog RF front-end and the software that implements the protocol.

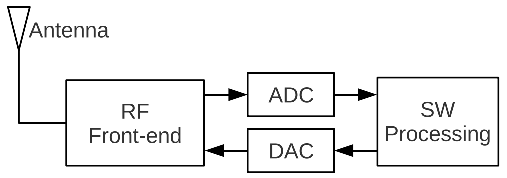

Figure 1. Software-defined radio general architecture

A software implementation of most protocols requires a big amount of processing power, and this usually goes against other design metrics of embedded systems, like size, cost, and energy consumption. However, the use of general purpose systems is very popular for implementing SDRs. Frameworks like GNU Radio [7] can be used to implement SDRs easily from high-level models using common PCs. Nevertheless, the predominance of the other metrics mentioned above may not allow the use of a powerful general purpose hardware platform, like a high-end PC, to implement the system.

Because of this hindrance, alternative solutions must be used in order to allow for the use of SDR on embedded systems. Usually, developers use more specific devices such as DSPs, GPPs with SIMD (Single Instruction, Multiple Data) co-processors and even programmable logic devices (PLD) like FPGAs (Field Programmable Gate Array), that are used to move parts from the software layer to the hardware layer in order to obtain better efficiency. However, the use of these hybrid systems may present high project risks due to the difficulties in translating a high-level model of the system to an implementation [14].

A careless design may present many complications, especially if it is going to be modified in the future to support new protocols.

So, in order to overcome these issues, we are proposing a new architecture for the implementation of SDRs on embedded systems. Following the *Application-driven Embedded System Design* (ADESD) methodology [5] and using the concept of *hybrid HW/SW components* [11], the architecture proposed in this work allows for the efficient and intuitive implementation of SDRs in FPGAs. This is achieved by mapping parts of a high-level functional model of the SDR directly to components that can migrate between the hardware and software domains in a transparent way.

The rest of the paper is organized as follows: section 2 talks about the SDR implementation approaches and the advantages and disadvantages of each approach. Section 3 introduces the concepts used to develop the proposed architecture described in section 4. Section 5 presents the architecture evaluation and results. The section 6 shows our conclusions.

2. SDR implementation approaches

There are many ways to implement SDRs. Overall, these implementations can be generalized into three basic approaches: using *general purpose processors*, using *programmable signal processing hardware*, and using *dedicated hardware*.

2.1. General purpose processors

The general purpose processor approach consists of the general SDR architecture. In this model, all the processing is done using a GPP, achieving the highest flexibility and ease in development. However, using general purpose hardware usually increases the costs of the system and the overhead imposed by general purpose operating systems can forbid the efficient implementation of time-strict protocols [12].

The GNU Radio [7] framework provides means of implementing SDRs on common PCs and is an example of an architecture that follows this approach. It offers a library of signal processing blocks and means to connect them to build an SDR. On GNU Radio the radios physical layer is abstracted as a flow graph where the nodes represent processing blocks and the edges the data flow between them.

2.2. Programmable signal processing hardware

In the second approach, the processing is done by programmable devices designed for specific functions. These devices includes: *digital signal processors* (DSP), GPPs with SIMD co-processors, and devices for specific functions like dedicated *digital up/down converters* (DUC/DDC). The hardware project in this approach tends to be more complicated and, typically, an FPGA is used to route the signals in the processing chain among the possible paths. Most of the architectures proposed in the last few years using this approach [6, 10, 15, 17] focus on the use of GPPs along with a set of co-processors optimized for fixed-point operations with both scalar and vector data. This way, they are able to achieve a good trade-off between performance and the other metrics listed in previous sections.

The main disadvantage of these architectures is the complexity of the software implementation. In SODA [10], for example, the synchronization between the GPP and the SIMD co-processors, and the optimization of the signal processing functions, need to be made manually in assembly code.

2.3. Dedicated hardware

In the dedicated hardware approach, a large part of the processing chain has a fixed implementation on PLDs. This approach may look like the traditional radio way, but, since the hardware can be reconfigured, the radio can be considered an SDR. Of all approaches, this one yields the best trade-off between hardware cost, size, and performance [16]. However, the unfriendly environment of developing the signal processing functions and the lack of software support for controlling the data flow in the hardware may present high project risks and time-to-market [2].

Ideally, architectures that keep the high level of abstraction for SDR developers should be used, for example the, GNU Radio. However, as it was explained earlier, most of the applications cannot be implemented using PCs. Architectures that use dedicated hardware may meet the application requirements, but offer many difficulties in translating a high-level model of the SDR to an implementation. Therefore, SDR developers should try to follow an approach that combines both the good trade-off of programmable hardware and the flexibility of architectures like GNU Radio.

3. Application-driven Embedded System Design

The ADESD (*Application-driven Embedded System Design*) [5] methodology provides ways of developing systems dedicated to a certain application by using preexisting components. These components are generated by a domain engineering process that identifies entities that are part of a certain application, and organizes them into families of reusable components that abstract these entities. The component dependencies of their original scenario are reduced using *aspect-oriented programming* [8] concepts. These concepts provide means of identifying scenario variations and modeling them as aspects. These aspects can be applied to abstract components in order to generate an implementation appropriate for a certain scenario. Figure 2 shows the relationship between the main entities obtained from the ADESD domain decomposition: families of scenario-independent abstractions, scenario adapters, and inflated interfaces.

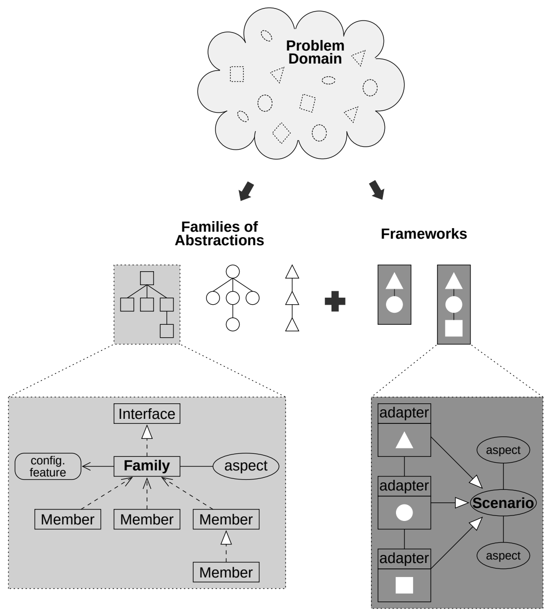

Figure 2. Overview of domain decomposition through ADESD

EPOS (*Embedded Parallel Operating System*) is an operating system based on the ADESD methodology. It aims to automate the development of dedicated computing systems, so that developers can concentrate on what really matters: their applications. EPOS features a set of tools to select, adapt, and plug components into an application-specific framework, thus enabling the automatic generation of an application-oriented system instance.

3.1. Hybrid hardware/software components

Hardware mediators are a particular kind of component that are responsible for keeping the high-level abstractions independent of the hardware platform. These components can be seen as part of an abstraction layer for configurable hardware. When combined with IPs on a PLD, the hardware mediators provide a customizable software–hardware interface that could lead to a dedicated platform, that is, a platform with only the necessary features to support the application.

On ADESD, the hardware mediators are implemented using *generative programming* [4] techniques, adapting the hardware interface to the interface required by the system instead of creating a hardware abstraction layer. The concept of *hardware/software components* [11] was elaborated on by the concept of hardware mediators. The idea of hybrid components emerges from the fact that different mediators can exists for the same hardware component, each one designed for different purposes (e.g. changing the trade-off between performance and energy consumption). Each component aggregates mediators for its many implementations, which could be in hardware, software or both. According to system requirements of cost, performance, energy, etc., any one of these implementations can be selected without any change to the higher system layers that use the component.

4. An architecture for SDR implementation

As discussed in the previous sections, to achieve the best trade-off between the most common embedded system design metrics, SDR should be implemented using the dedicated hardware approach. However, this approach may present high project risks and restrict SDR flexibility. Ideally, an architecture like GNU Radio should be used, providing the high-level implementable abstraction of an SDR. But, as stated before, GNU Radio is not suitable for embedded systems. Architectures like SODA may meet the application requirements, but offer difficulties in converting a high-level model to an implementation.

Given the difficulties shown in the previous sections, this paper proposes a new architecture for SDR development in embedded systems. This new architecture uses the ADESD methodology and hybrid hardware/software components to enable the implementation of SDRs using a dedicated hardware approach. A framework allows the easy translation of a high level model of the SDR to an implementation.

4.1. The proposed architecture

Our architecture is based on the *Synchronous Data Flow* (SDF) model [9] that is a very common model for designing digital signal processing applications. In this model, the SDR processing chain is abstracted as a flow graph, where the nodes represent processing blocks and the edges represent the data flow between the blocks. Figure 3 presents an overview of the proposed architecture. The processing blocks have their functions implemented using hybrid components, so the processing blocks can be in hardware, software, or both, in a way which is transparent to the SDR developers. The blocks are connected through FIFO channels that are allocated by the architecture's framework in hardware or software, depending on the implementation of the blocks that it connects. An entity called *Flow Controller* controls the data flow between the blocks.

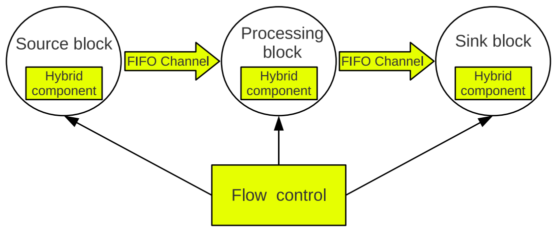

Figure 3. Overview of the proposed architecture

As depicted in figure 3, the architecture defines three different kinds of blocks:

Source blocks are responsible for the insertion of data into the flow graph. They usually abstract input devices like RF front-ends and ADCs.

Processing blocks are responsible for the signal processing.

Sink blocks consume the data from the flow graph. They can abstract output devices like a DAC and an RF front-end, or provide a callback mechanism for higher protocol layers.

Figure 4 shows the interface for the three kinds of blocks. Each block must provide information about it and implement its functionality. The processing blocks, for example, define the methods *do_processing* and *block_info*. The *do_processing* method is where the signal processing is implemented. The *block_info* method returns some information required by the flow controller mechanism: the number of inputs and outputs and the block's input/output rate; that is, the number of data elements consumed and generated during each execution of *do_processing*. This information is used by the flow controller to correctly connect the blocks and calculate the FIFO's size. The interfaces for the source and sink blocks are very similar to the processing blocks, except that they don't have inputs and outputs, respectively.

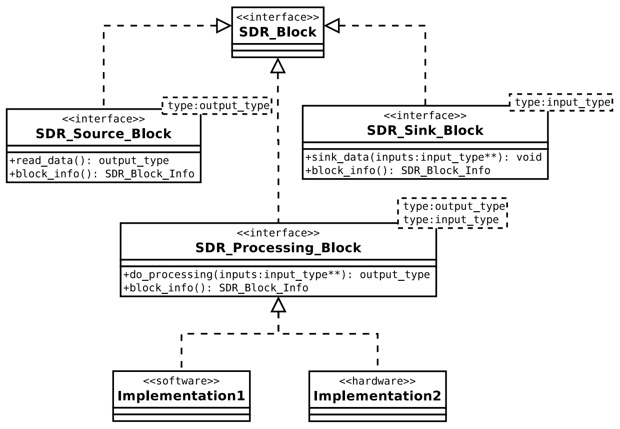

Figure 4. Source, processing, and sink blocks interface

When the blocks are implemented on hardware, it is not necessary to provide an implementation of the block functionality on the software interface. The interface implementation should just provide additional information about how the block is connected to the architecture's hardware layer. The support for hardware blocks will be explained in more detail in section 4.3.

4.2. Flow controller mechanism

The flow controller is responsible for connecting the blocks and controlling the data flow between them. When the flow controller connects two blocks, it first checks if the source block output and the destination block input match. Then, it creates a FIFO channel to connect the blocks. The size of the FIFO in the channel is defined by the following equation:

$$FIFO_{size} = (Blk_0^{outputrate} + Blk_1^{inputrate}).\alpha \quad (1)$$

Where an output of Blk_0 is being connected to an input of Blk_1, $Blk_0^{outputrate}$ is the number of data elements generated upon each execution of Blk_0, and $Blk_1^{inputrate}$ is the number of data elements consumed in each execution of Blk_1. α is a safety factor that should be set according to the memory and performance characteristics of the system, in order to make sure the FIFO will be big enough to handle all the data generated by Blk_0 until it can be consumed by Blk_1.

The control of the data flow between the blocks is accomplished by creating a thread for each block, where its function is executed. The threads are synchronized using semaphores associated with the FIFO channels. At first, the threads remain locked on to the semaphores associated with the channels connected to the block's inputs. Each time an element is added to a channel, the *v()* method of its associated semaphore is called, unlocking the threads that consume the data from the channels. This mechanism applies only to software blocks. The control of the data flow between hardware blocks will be explained later.

The pseudo-code in figure 5 shows this behavior for a processing block. $In_{0..n}$ and $Out_{0..n}$ are the channels associated with the block's inputs and outputs, respectively. The semaphore is encapsulated by the FIFO channel and the *wait* and *signal* methods are related to the *p* and *v* operations of the semaphore.

```
Processing block loop :
     In0.wait()
     ...
     Inn.wait()
     If there are enough inputs:
          Consume inputs
          Do processing
          For each output generated:
               Write outputs to Out0
               ...
               Write outputs to Outn
               Out0.signal()
               ....
               Outn.signal()
```

Figure 5. Processing block behavior

4.3. Hardware support

In order to provide support for hardware blocks and keep the flexibility of how the blocks are connected in a transparent way, we have also designed the FIFO channels as hybrid components, as it is shown in figure 6.

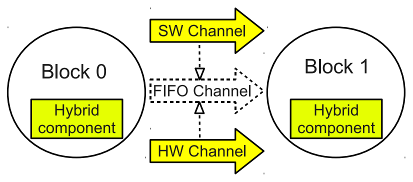

Figure 6. FIFO channels behave like hybrid components

The deployment of HW FIFO channels is supported by the flow controller hardware structure shown in figure 7. This structure consists of a set of read ports, write ports and FIFOs, where the connection between these elements can be defined by software-controlled configuration registers.

All the blocks that are implemented as hardware have their inputs connected to the structure's read ports, and their outputs connected to the write ports. When these blocks are connected, the flow controller mechanism uses the information provided by the block's software interface to define which port must be connected to which FIFO. Since the HW FIFOs must have a fixed size, the interconnect structure allows FIFOs to be interconnected among them. This way, when two blocks are connected, it is possible to allocate a chain of FIFOs between them, in a way in which the total size of the chain is bigger than or equal to the required FIFO size, as defined on section 4.2

The communication between software and hardware blocks is achieved through the *Block SW Interface* shown in figure 7. It behaves like a wrapper between the system bus and the interconnect structure stream interface. When a block in hardware is connected to a block in software, its respective ports are connected to ports associated with the *Block SW Interface*.

Also, since the blocks and the FIFO channels are hybrid components, the flow controller tries to control all the blocks and access the FIFOs in the same way. But, for the HW blocks and FIFO channels, the sequence of $wait \rightarrow processing \rightarrow signal$ operations shown in figure 5 are handled directly in hardware through signals controlled by the interconnect structure. In this case, the operations shown in figure 5 are actually empty, resulting in zero software overhead for the blocks in hardware.

Figure 9 shows how we implemented the interconnect structure of figure 7. We implemented a simplified butterfly fat tree NoC architecture [13], with optimizations for our application. It consists of a matrix of FIFOs where each FIFO input is connected to each input port, and each output port is connected to each FIFO output. Each FIFO output is also connected to the input of the FIFO in the next column on the same line. With this interconnect scheme we can provide a wide range of possible allocation for each input/output port, while keeping the use of PLDs resources by interconnect in a reasonable level.

This interconnect structure was designed to be fully parameterizable. We are able to choose the size of the FIFO matrix in both dimensions, as well as the FIFOs width and depth, so the synthesized hardware can fit in the chosen device and meet the application requirements.

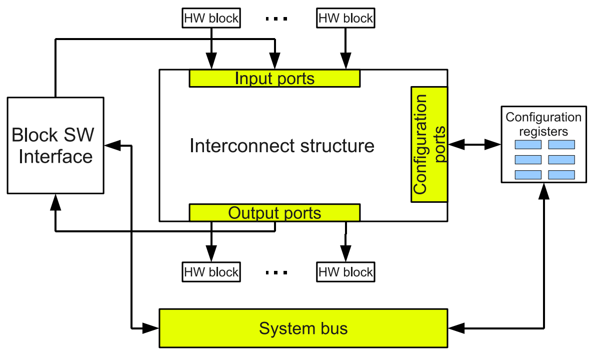

Figure 7. Flow controller structure HW support

5. Evaluation and results

5.1. Evaluation setup

In order to evaluate the proposed architecture, we have dpne tests to measure the overhead imposed on the data flow by the architecture. We have implemented the three structures shown in figure 8 to evaluate the increase in overhead in terms of the number of blocks in serial, the number of blocks in parallel and the number of inputs/outputs of a block, respectively. This way we can determine how the architecture control structures are going to behave for all possible SDR data flow structures.

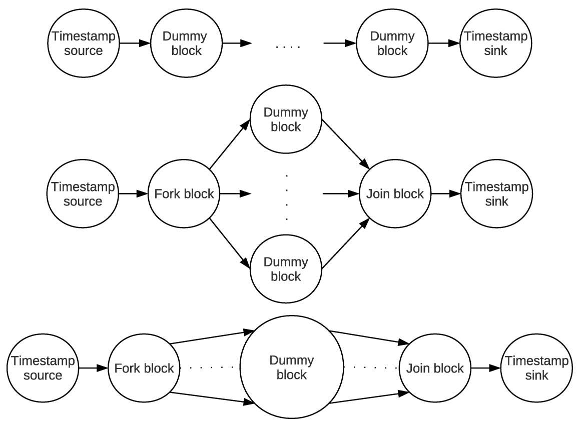

Figure 8. SDRs implemented for the overhead evaluation

A source block generates samples that contain timestamps of when they were generated. The *dummy blocks* are empty blocks that just propagate their inputs to their outputs. When the samples arrive at the sink block, the timestamps are compared with the current time, obtaining

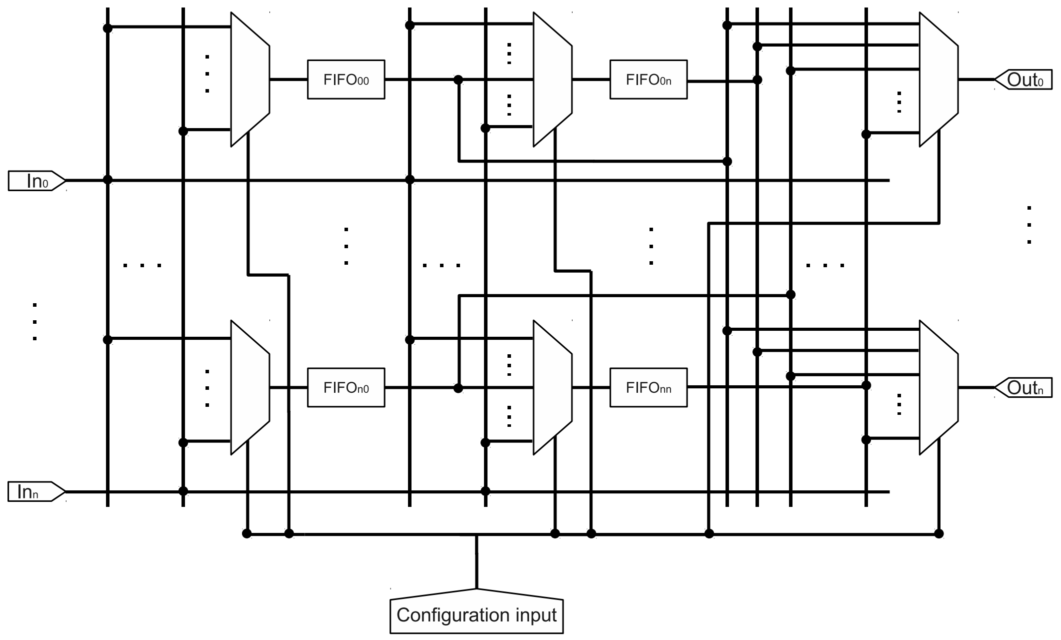

Figure 9. Overview of the flow controller HW interconnect structure

Table 1. Configuration parameters used on the experiments

Parameter	Value
Synthesis tool	ISE/EDK 10.1
Compiler	GCC 4.0.2
FPGA clock	100 MHz
Microprocessor clock	100 MHz
Num. of input ports	32
Num. of output ports	32
HW FIFO size	8 bits wide with 16 elements
Num. of FIFOs	64

the time the sample took to go through the block chain. Since the blocks are empty, this resulting time is the overhead imposed by the architecture on the data flow.

5.2. System implementation and configuration

To evaluate these three structures, we have implemented the proposed architecture on the Xilinx's Virtex-4 ML403 Embedded Platform [1]. The ML403 features a Virtex-4 XC4VFX12 FPGA with an embedded PowerPC 405 microprocessor. In order to use the same hardware configuration, we have synthesized the hardware with all of the necessary blocks for each experiment. Table 1 shows the parameters used in the experiment's setup. The last four parameters are related to the configuration of the interconnect structure in hardware.

Table 2 shows the resource consumption of the generated hardware. Separate results are shown for the synthesized architecture interconnect structure along with the HW processing blocks, and for the system IPs generated by EDK (internal memory, memory controller, interruption controller, UART, etc). Due to the lack of memory blocks available on the device, we choose to implement the FIFOs using the SRL16 capabilities to convert a 4-input LUT into a 16-bit shift register. This feature not only allows us to save memory blocks, but also yields high performance and low cost FIFO implementation [3].

Table 2. Amount of HW resources used by the synthesized structures

Resource	Our IPs	EDK IPs	Full System
4-input LUTs	13%	35%	49%
Slice Flip Flops	67%	31%	98%
Occupied Slices	49%	55%	99%
RAM blocks	0%	63%	63%
Max. frequency	167 MHz	109 MHz	107 MHz

The application along with the EPOS operating system running on the PowerPC processor resulted in a memory footprint of 47632 bytes of code and 288 bytes of static data.

5.3. Results

Figures 10 and 11 show the average and the coefficient of variation of the latency for each of the three data flow structures presented earlier. When using only software blocks, the overhead grows linearly in relation to the increase in the number of blocks and the number of inputs and outputs for all structures. The coefficient of variation remained low in all configurations, and the lowest values for the multiple input/output dummy block configurations were due to the constant number of threads in the system. When using only HW blocks, the latency was about four orders of magnitude lower than when using software blocks and, as expected, except for the serial block configuration, the latency remained constant regardless of the size of the structure, due to parallelism in the hardware operations. There is also a null coefficient of variation in the hardware operations.

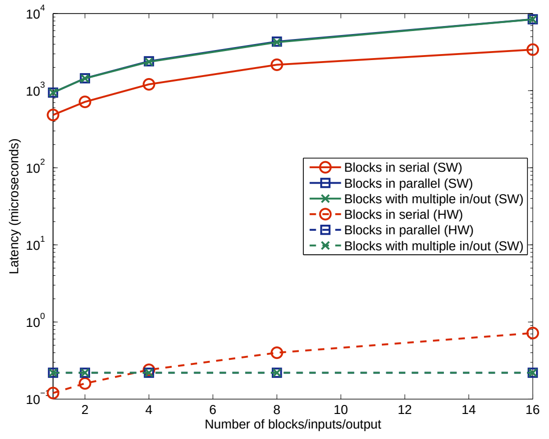

Figure 10. Average latency for blocks in serial, in parallel and with multiple input/output

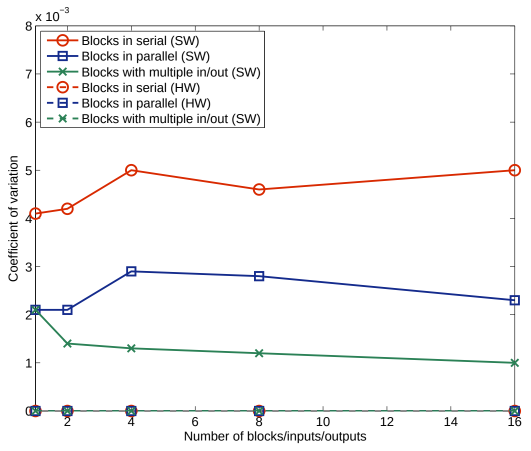

Figure 11. Coefficient of variation of the proposed architecture latency

We also compared the overhead of our architecture to GNU Radio. For this comparison, we implemented the same data flow structures described in section 5.1 using GNU Radio running over a Linux operating system in a PC. Our architecture and EPOS were compiled for the IA32 architecture only with the software blocks and evaluated in the same system. For the GNU Radio experiment, we used GNU Radio 3.2.2 running on a Linux kernel 2.6.28. Figures 12 and 13 show the results of the serial blocks data flow structure.

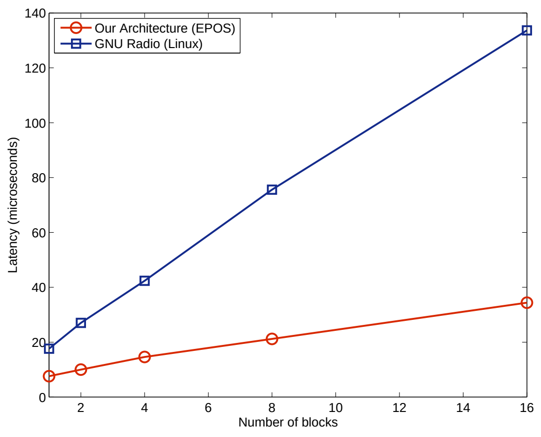

Figure 12. Average latency of the proposed architecture VS GNU Radio on the serial blocks data flow structure

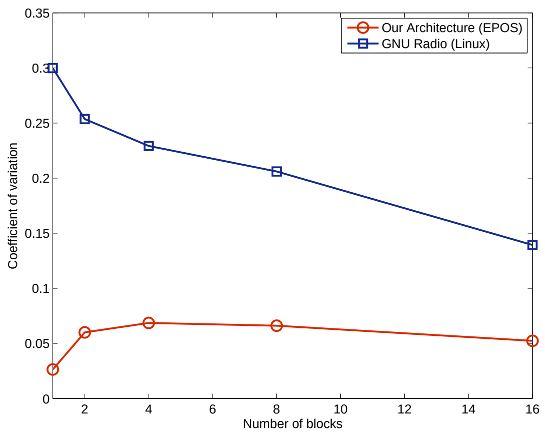

Figure 13. Coefficient of variation of the proposed architecture latency VS GNU Radio on the serial blocks data flow structure

In order to avoid interference from other Linux processes on the GNU Radio experiment and provide a fairer comparison, the application was executed using the *enable real-time scheduling* option available on GNU Radio, which gives the highest priority to the GNU Radio process. Nevertheless, our architecture performance surpasses GNU Radio between 2 and 4 times, and this difference increases as the number of blocks in the processing chain increases. Figure 13 also shows that we are able to achieve smaller latency variations as well.

5.4. Discussion

The results show that our architecture yields superior performance than an equivalent, in terms of abstraction level, commonly used architecture. Also, the virtually null variation obtained when hardware blocks are used in a dedicated platform (ML403) is especially important when implementing time-strict protocols, like TDMA based protocols. This is not the case with GNU Radio, for example. On GNU Radio, the use of any hardware device to get or sink data from/to the environment requires Linux drivers whose performance is mostly limited by the kernel's abstraction layer. A previous work [12] shows that, due to the Linux kernel overhead, the standard deviation of the time a sample takes to get to the processing chain after being generated at the RF Front-end is higher than the average time.

6. Conclusion

In this paper we presented an analysis of SDR implementation approaches and proposed a new architecture for SDR development. This new architecture enables SDR development using PLDs by directly mapping components of a high-level model to implementable components. The results show that the architecture imposes a very low and deterministic overhead along the SDR processing chain.

References

[1] Virtex-4 ML403 Embedded Platform, 2010. [Online; accessed 09-March-2010].

[2] J. A. Bower, W. N. Cho, and W. Luk. Unifying FPGA Hardware Development. In *ICFPT 2007. International Conference on Field-Programmable Technology, 2007*, pages 113–120, 2007.

[3] K. Chapman. Saving Costs with the SRL16E. Technical report, Xilinx, May 2008.

[4] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[5] A. A. Fröhlich. *Application-Oriented Operating Systems*. PhD thesis, Technical University of Berlin, Berlin, 2001.

[6] J. Glossner, E. Hokenek, and M. Moudgill. The Sandbridge Sandblaster Communications Processor. In *3rd Workshop on Application Specific Processors*, pages 53–58, 2004.

[7] GNU FSF project. The GNU Radio, 2009. [Online; accessed 08-November-2009].

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, pages 220–242, Jyväskylä, Finland, June 1997.

[9] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, C-36:24–35, 1987.

[10] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. SODA: A Low-power Architecture For Software Radio. In *ISCA '06: 33rd International Symposium on Computer Architecture*, pages 89–101, 2006.

[11] H. Marcondes and A. A. Fröhlich. On Hybrid Hw/Sw Components for Embedded System Design. In *Proceedings of the 17th IFAC World Congress*, pages 9290–9295, Seoul, Korea, 2008.

[12] G. Nychis, T. Hottelier, Z. Yang, S. Seshan, and P. Steenkiste. Enabling MAC Protocols Implementation on Software-defined Radios. In *Networked Systems Design and Implementation*, 2009.

[13] P. P. Pande, C. Grecu, A. Ivanov, and R. Saleh. Design of a switch for network on chip applications. In *ISCAS '03. Proceedings of the 2003 International Symposium on Circuits and Systems*, volume 5, pages V217–V220, 203.

[14] I. R. Quadri, S. Meftali, and J.-L. Dekeyser. High level modeling of dynamic reconfigurable FPGAs. *International Journal of Reconfigurable Computing*, 2009:1–15, 2009.

[15] Steven Kelem, Brian Box, Stephen Wasson, Robert Plunkett, Joseph Hassoun, and Chris Phillips. An Elemental Computing Architecture for SD Radio. In *SDR '07: 2007 Software Defined Radio Technical Conference*, 2007.

[16] N. Sulaiman, Z. A. Obaid, M. H. Marhaban, and M. N. Hamidon. Design and Implementation of FPGA-Based Systems - A Review. *Australian Journal of Basic and Applied Sciences*, 3(4):3575–3596, 2009.

[17] 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: Journal on Applied Signal Processing*, 2005:2613–2625, 2005.