Software-Defined Wireless Communication Systems for Heterogeneous Architectures

David Volz
TU Darmstadt
Germany
volz@esa.tu-darmstadt.de

Andreas Koch
TU Darmstadt
Germany
koch@esa.tu-darmstadt.de

Bastian Bloessl
TU Darmstadt
Germany
mail@bastibl.net

ABSTRACT

Future cellular networks will be programmable and increasingly software-defined with APIs to hook into the communication stack closer and closer to the physical layer. This allows operators, for example, to plug-in third-party machine learning algorithms to optimize performance. At the same time, this flexibility implies that compute resources cannot be provisioned statically but have to be distributed dynamically during runtime. While the hardware platforms for such systems are available, we lack suitable software frameworks that help to realize such systems. In this demonstration, we present two Open Source projects that fill this gap: *FutureSDR*, a portable real-time signal processing framework with native support for accelerators (like GPUs and FGPAs); and *IPEC*, which enables fully automatic composition of multi-accelerator FPGA designs. We believe that these tools – especially in combination with each other – can provide the base for building research prototypes and allow experimentation with software-defined wireless communication systems.

1 INTRODUCTION

With the trend towards softwarized, disaggregated Radio Access Networks (RANs) [2], Software Defined Radio (SDR) will become a key technology for telecommunication networks. This goes beyond traditional applications, where an SDR framework interacts directly with the radio but extends to (near) real-time control loops in O-RAN applications [1], which require high-performance stream-data processing systems. Yet, while SDR hardware made great leaps forward, the required programming frameworks did not match this progress, which is why building software-defined wireless communication systems remains a challenge. General-purpose SDR frameworks like GNU Radio [3] have conceptual limitations that cannot easily be overcome and, thus, lose relevance for complex state-of-the-art applications with non-trivial compute and timing requirements.

2 FUTURESDR

FutureSDR[1] is designed and implemented from scratch, taking a fresh look at many long-standing issues. Yet, the main abstractions of FutureSDR are familiar to people working in the application domain. It uses *Blocks* that implement a step in the signal processing chain, e.g., a filter, an arithmetic operation, or a synchronizer. The blocks have input and output ports that are connected to form a *Flowgraph*. The ports are either message ports (for asynchronous message passing) or stream ports (for streams of samples). In addition, FutureSDR uses the concept of a *Runtime* that has a *Scheduler* associated with it and that is able to run flowgraphs.

To achieve this, FutureSDR is based on a runtime for asynchronous tasks, which eases I/O integration and allows full control over scheduling decisions. This can be regarded as a switch from threads to tasks and, hence, from parallelism to concurrency. Furthermore, using tasks instead of threads allows us to control scheduling, since the asynchronous task executor is implemented in user space as part of the application. Threads, in turn, are managed by the operating system scheduler with limited control by the application. Custom asynchronous executors can exploit knowledge of the application's flowgraph topology to make better decisions. Plugging in different schedulers, we can optimize for architectures, applications, or configure the trade-off between metrics like throughput and latency.

Furthermore, FutureSDR has native support for heterogeneous systems. This refers to three dimensions: heterogeneity in terms of platforms (e.g., Android, Windows, Linux, the web), architectures (e.g., x86, ARM, and WebAssembly), and compute (e.g., CPU, FPGA, and GPU). Support for this wide range of systems is not added as an afterthought but part of the core design of the framework.

[1] https://www.futuresdr.org/

ACM MobiCom '23, October 2–6, 2023, Madrid, Spain

ACM ISBN 978-1-4503-9990-6/23/10.
https://doi.org/10.1145/3570361.3614084

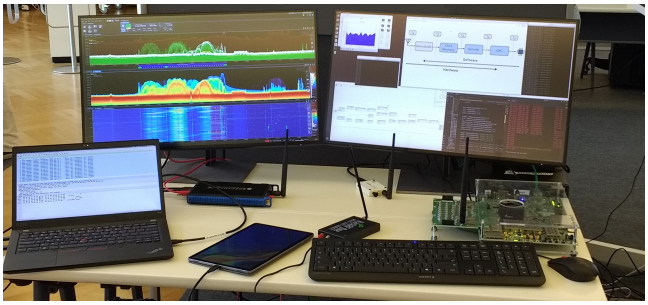

Figure 1: Photo of demonstration setup.

3 IPEC

Inter-Processing Element Communication (IPEC) [4] is a framework that simplifies the development of complex System on Chips (SoCs) with heterogeneous Processing Elements (PEs), together with their supporting interconnect structures and memory hierarchies. As FPGAs continue to grow in size, they can hold a large number of complex PEs. Tools for building individual PEs exist and have improved significantly in recent years (e.g., powerful hardware construction languages, such as Chisel or Bluespec). However, building an SoC composed of many interconnected PEs remains a challenge that IPEC tries to solve.

IPEC views the communication between PEs and memory as a data-flow graph that can be described in a Python-based Domain-Specific Language (DSL). For the communication between PEs, it supports multiple protocols, such as AXI4, AXI4Lite, and AXI4Stream, as well as more abstract stream-based operations, such as reductions or atomic operations for inter-PE-synchronization. Furthermore, IPEC allows to combine PEs for stream-based and task-based processing, as well as software-programmable PEs for those parts of an application where flexibility is more important than raw performance. The user can choose from a library of generic PEs provided by IPEC (e.g., various soft-core processors), or can include their own PEs, which are imported in the industry standard IP-XACT format. IPEC reads the data-flow description and generates a complete SoC, containing the required PEs and interconnect structures, which can then be synthesized and programmed onto an FPGA.

IPEC simplifies testing hardware designs for wireless communications, allowing the user to input pre-recorded IQ traces or samples that are generated with existing software implementations. This enables detection of functional errors in the implementation before having to synthesize the entire system down to hardware.

4 DEMONSTRATION SETUP

The demonstration setup is shown in Figure 1. To highlight the main features of our frameworks, we have the same

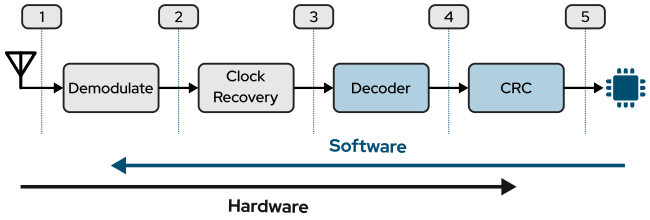

Figure 2: Different stages in the protocol implementation.

FutureSDR receiver running on three very different platforms: (1) on a normal laptop, interfacing an Aaronia Spectran v6 SDR, (2) on a web browser, compiled to WebAssembly and interfacing a HackRF SDR through WebUSB, (3) on a AMD/Xilinx RFSoC ZCU111 evaluation board.

Figure 2 shows a high-level overview of the receiver structure. The same protocol is implemented both in software (using FutureSDR) and in hardware (using IPEC). Since both implementations have the same structure, we can configure during runtime after which decoding stage to switch from FPGA to CPU processing. When selecting the third split, for example, IQ-samples from the RFSoCs ADCs are processed in the demodulation and clock recovery stage in hardware. The result is then streamed into DRAM, where the software implementation takes over and processes the remaining two stages. At our booth, the visitor can set this dynamically and observe the change in CPU load on the processor.

5 CONCLUSION

We present two Open Source projects that can be used to build portable, reconfigurable, real-time signal processing systems. To this end, we implement the same receiver on various platforms, including the Xilinx ZCU111, where it is utilizing both the FPGA and the CPU cores of the RFSoC. Both the software and hardware implementation cover the full protocol, with the ability to switch between intermediate stages during runtime. Together, this highlights two important features: (1) We show the portability of FutureSDR, having the same receiver running on very different platforms (PC, WebAssembly, RFSoC). (2) We show that the software implementation is capable of offloading different parts of the computation depending on the situation. Releasing the software as Open Source, we hope that it will prove useful to build research prototypes for software-defined wireless communication systems. [2]

[2] *Acknowledgements:* This work has been co-funded by the LOEWE initiative (Hessen, Germany) within the *emergenCITY* center, as well as the German Research Foundation (DFG) in the Collaborative Research Center (SFB) 1053 MAKI. The authors acknowledge the financial support by the Federal Ministry of Education and Research of Germany in the project "Open6GHub" (grant number: 16KISK014).

REFERENCES

[1] Salvatore D'Oro, Michele Polese, Leonardo Bonati, Hai Cheng, and Tommaso Melodia. 2022. dApps: Distributed Applications for Real-Time Inference and Control in O-RAN. *IEEE Communications Magazine* 60, 11 (2022), 52–58. https://doi.org/10.1109/MCOM.002.2200079

[2] Michele Polese, Leonardo Bonati, Salvatore D'Oro, Stefano Basagni, and Tommaso Melodia. 2023. Understanding O-RAN: Architecture, Interfaces, Algorithms, Security, and Research Challenges. *IEEE Communications Surveys & Tutorials* 25, 2 (2023), 1376–1411. https://doi.org/10.1109/COMST.2023.3239220

[3] Thomas W. Rondeau. 2015. On the GNU Radio Ecosystem. In *Opportunistic Spectrum Sharing and White Space Access: The Practical Reality*, Oliver Holland, Hanna Bogucka, and Arturas Medeisis (Eds.). Wiley. https://doi.org/10.1002/9781119057246.ch2

[4] David Volz, Christoph Spang, and Andreas Koch. 2022. IPEC: Open-Source Design Automation for Inter-Processing Element Communication. In *Applied Reconfigurable Computing. Architectures, Tools, and Applications*. Springer International Publishing. https://doi.org/10.1007/978-3-031-19983-7_10