PYNQ Environment on Versal Devices

1st Nemanja Filipović
School of Electrical Engineering, University of Belgrade and Tannera LLC
Belgrade, Serbia
email: nemanja@tannera.io

2nd Dragomir El Mezeni
School of Electrical Engineering University of Belgrade
Belgrade, Serbia
email: elmezeni@etf.bg.ac.rs

3rd Vladimir L. Petrović
School of Electrical Engineering, University of Belgrade and Tannera LLC
Belgrade, Serbia
email: petrovicv@etf.bg.ac.rs

***Abstract*—Python productivity for Zynq (PYNQ) is a Linux environment targeted for use on Zynq UltraScale+ and Zynq-7000 platforms from AMD. Its main component is a Python library that facilitates writing drivers, performing direct memory access (DMA) transfers and executing other complex tasks directly from the Python environment. This enables the use of a high-level programming language with a rich library set to control the implemented hardware design directly. However, the newest architecture of AMD all-programmable systems on chip (SoCs), Versal does not support PYNQ. It represents a paradigm shift, with a tighter coupling between the field-programmable gate array (FPGA) fabric, the processing system, and the newly introduced artificial intelligence (AI) engines. This makes the straightforward porting of the PYNQ environment to the new hardware impossible. Through the use of specific implementation settings and modifications to the PYNQ library, the work from this paper successfully ported much of the PYNQ functionality to the new platform. These modifications were tested on a VCK190 board, featuring a Versal Core XCVC1902 ACAP.**

***Index Terms*—PYNQ, FPGA, Versal ACAP, Zynq UltraScale+, MPSoC, Python**

I. Introduction

The newest FPGA device family from AMD (formerly Xilinx) is the Versal series [1]. Devices from that series introduced the new AI cores, the new Network on Chip (NoC), upgraded ARM processing system, and improved connectivity. The top-level architecture of the devices from the family is shown in Fig. 1.

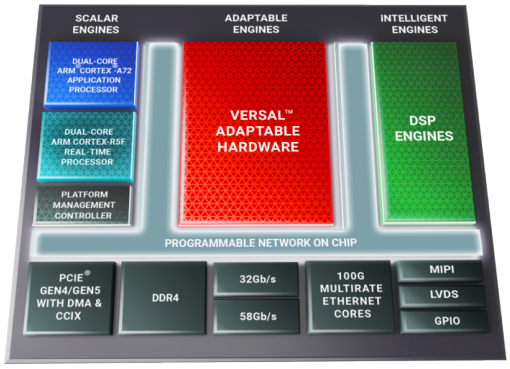

Figure 1. Illustration of the Versal ACAP structure provided by AMD [2]

The new programmable SoC type is now called an ACAP (Adaptive Compute Acceleration Platform) and it represents a paradigm shift toward a tight integration between all of the components through the NoC [3]. The new AI engines enable the implementation of AI and signal processing tasks on a grid of very long instruction word (VLIW) processors with integrated memory and direct memory access (DMA) controllers [4]. This means that a single solution can use the FPGA fabric, the distributed processing of the AI engines, and the classic real-time or application software running on the ARM cores. It is possible to use Vitis Unified Software Platform tools for HLS [4], i.e. to write the code for the entire system in the same C-like environment, and to optimize it using pragmas. The user can still maintain control and follow the earlier design flow with the register transfer level (RTL) design being written in Verilog or VHDL. Currently, the user needs to follow the entire ACAP design workflow in order to access such a design from software. This workflow can be cumbersome in the prototyping stage of the design, and the process can become convoluted quickly.

To increase productivity and reduce the time between having an RTL design and running it on actual silicon, AMD/Xilinx introduced Python Productivity for Zynq (PYNQ) [5], which supports previous generations of their SoCs. This environment runs on an Ubuntu distribution of Linux, featuring the PYNQ Python library and the Jupyter Lab server. All of the software components are running onboard the application processors of the device. Jupyter Lab enables direct access to the device, with an interactive Python interpreter enabling code execution directly from the web browser. For programming the FPGA fabric of the SoC, a bitstream file generated by the Vivado implementation tool is used. Direct programming of the FPGA fabric is possible directly from the Python interpreter. PYNQ introduces a Python class called `Overlay`, which provides the abstraction of the entire FPGA design and its components. Its constructor takes as an argument the bitstream file, and, when called, programs the FPGA programmable logic (PL).

To facilitate access to the peripherals in the PL, the PYNQ library also handles their memory map. This is done through a hardware handoff (`.hwh`) file generated using the IP integrator flow in Vivado. The `.hwh` file is an XML-formatted file containing design information, including address offsets, register maps, and types of IP cores connected to the processing system using the AXI4 memory-mapped interface. Upon construction, the `Overlay` class parses the `.hwh` file which is bundled with the bitstream and identifies the peripherals that the

This publication has been produced with the financial assistance of the European Union through the Innovation Fund of the Republic of Serbia under grant ID 50427, entitled "Accelerate 5G: Hardware accelerator IP cores for 5G infrastructure". The contents of this document are the sole responsibility of the authors and can under no circumstances be regarded as a reflection of the position of the European Union.

processing system (PS), containing ARM CPU cores, can access. It then compares the type of peripheral (vendor, library, name, and version (VLNV) string from Vivado) with the list of available drivers and instantiates either a dedicated driver or a generic driver. The generic driver enables access to the registers of the peripheral, through `read` and `write` methods. Those methods take the address of the register relative to the component's base address, while the base address is obtained from the parsed `.hwh`. Dedicated drivers for common IPs, such as DMA and general-purpose input/output (GPIO) controllers or RF Data converters in the case of RFSoCs are already delivered with PYNQ. The user can implement custom drivers easily, by defining a Python class derived from the `DefaultIP` class. For DMA controllers, PYNQ provides access to a library for the allocation of contiguous memory which is seamlessly integrated with both the DMA controller driver and the NumPy library array [6].

The described workflow enables the bring up of a possibly complex set of IPs, with memory-mapped control and streaming data interfaces with just several lines of Python code. If, for example, a streaming FFT IP core has been designed, it can be integrated into the PL with a DMA controller, and downloaded using PYNQ. The input data and the results can be generated and compared against a floating point FFT implementation from the NumPy library. The IPs can also be verified directly on the FPGA with minimal setup, using the `Pytest` library. Another benefit of such a workflow is that it enables remote operation of the board, over the Ethernet without the need for a dedicated PC running Vivado Hardware Manager [7]. This feature set vastly increases the efficiency and the ability to early test and verify RTL design, whether encompassing the entire system or merely a constituent component thereof. The PYNQ framework can also be used for the rapid prototyping of machine-learning tasks [8] or deployment of flexible neural network implementations [9].

However, besides all the advantages of the PYNQ workflow, it still has not been supported for the Versal devices. Moreover, the PYNQ porting on Versal is still not in AMD's development pipeline [10]. This provided the motivation to integrate the advantages of Versal ACAP devices with such a powerful environment. Therefore, this paper addresses this issue and provides a method for successfully porting PYNQ on Versal.

II. Vivado Synthesis and Implementation Flow for PYNQ

A. Versal-Specific Boot Sequence

Zynq (MPSoC and Zynq-7000) had a clear separation between FPGA fabric and its systems, meaning that the user could program the FPGA portion like it were another device, with userspace drivers. Zynq processing system (PS) from the designer's point of view is available as a monolithic hard IP core that communicates with the PL using AXI4 slave and master interfaces and does not expose any internal memory interfaces to the user.

However, the Versal ACAP architecture features much more tightly integrated components. The PS is integrated into the larger hard IP core, called the Control, Interfaces, and Processing System (CIPS), which, as a standalone component, is not capable of booting the operating system. The CIPS cannot operate without some programmable logic. Additionally, the only way to program the PL is through CIPS using the first-stage bootloader (FSBL). This causes issues when trying to implement a PYNQ workflow. Linux cannot boot if the PL is not programmed, whereas the PYNQ workflow is based on reprogramming the hardware in runtime. This is one of the main PYNQ benefits, enabling the user to quickly implement the design (without going into Vitis- or petalinux-specific tasks) and testing/integrating it immediately in a high-level Python environment.

To better understand why is it fundamentally necessary to program the PL before booting the OS, observe the minimum configuration needed for booting a Linux image in Fig. 2.

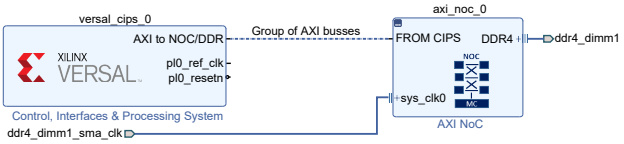

Figure 2. Minimum CIPS connections to boot Linux

The connection to the DDR goes through PL, i.e., through the NoC, which is configured together with the PL. NoC is a novel concept introduced in the Versal architecture, which represents a series of hard IP-like AXI4 and AXI4-Stream interconnects that are placed across the entire chip [11]. It also features DDR controllers that are used for connecting to the external RAM. This part is almost statically connected in the SoC fabric but still needs to be configured using FSBL before booting the application. Having in mind that this is a significant difference from the previous generation, AMD/Xilinx introduced the Classic SoC Boot mode in their Vivado software suite [12]. It leverages partial reconfiguration to program the minimum CIPS setup as a static design, while the "PL" design is kept in a dynamic region. The Classic SoC Boot mode provides a set of constraints for the dynamic region, such that the entire FPGA fabric is encompassed, and that the static region is always implemented and placed in the same way, allowing the user to run implementation in multiple projects. This feature will be used to run PYNQ, emulating the Zynq behavior.

B. The Classic SoC Boot Project

In order to facilitate the valid PYNQ workflow, it is necessary to create a Classic SoC Boot8 project in which user-logic interfaces are configured and exposed [13]. The static region of this base project will be used for booting the board. The most common PYNQ workflow requires the CIPS AXI master interface for accessing peripherals, as well as the DDR AXI slave interface for executing DMA transfers. The master interface of the CIPS is available directly, whereas the DDR slave interface is exposed through the NoC, meaning that the DMA controller drives the memory writes without going through the CIPS. Following the Classic SoC Boot constraints, the DDR AXI slave connection is exposed to user logic via the inter-NoC

interconnect (INI) interface. The static region block design with the the dynamic region interface is shown in Fig. 3.

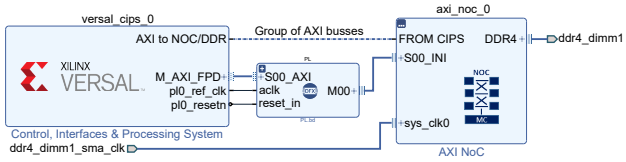

Figure 3. Block design of the static region with the exposed dynamic region (`PL`) interfaces

In the rest of the paper, an example design with a DMA loopback in the dynamic region will be used for demonstration and testing (Fig. 4). Even though it is simple, it demonstrates both the memory-mapped and the DMA access, which are core PYNQ functionalities.

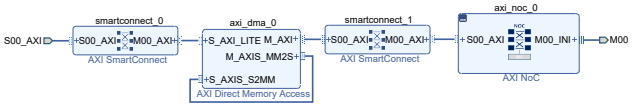

Figure 4. Example of a dynamic portion of the design

Examining the design further, the `smartconnect_0` is used for driving the memory-mapped peripherals and performing address decoding, while the `smartconnect_1` handles the DMA traffic. Bridge from an AXI4 interface to the INI interface is done through `axi_noc_0`.

Both the static and the dynamic region designs are synthesized and implemented using standard Vivado flows. The tool automatically applies pblock constraints on the design when the project's `classic_soc_boot` property is set. The output of the implementation tasks is used for generating the device images. Device image (`.pdi`) is the new file format used for the ACAP, equivalent to a bitstream file from previous architectures. The static region device image is used when booting the device and the operating system, while the dynamic region image is used from the PYNQ environment, in the same manner as the legacy bitstream.

III. PYNQ Image and Library Modifications

A. SD Card Image Compilation

The PYNQ environment has a well-documented procedure for generating a device image for a new board, that is not readily supported [14]. It requires modifications of the configuration files and good knowledge of the build process. This task is significantly broadened when changing the target platform to a Versal board.

As a starting point for achieving all functionalities described above, an open-source project called MLIR-AIR [15] is used. It is an example board image used for prototyping with the AI cores that are available on Versal platforms. Prototyping is done through the use of the AIR dialect of MLIR, which enables the description of the AI engine graph and its control. In order to fast-track their development, the authors of the MLIR-AIR project used the readily available PYNQ v2.7 image as a starting point. MLIR-AIR image supports, among others, the VCK190 board. However, because it was compiled starting from a stock PYNQ image, it features the installed PYNQ library that is not functional and also boots the board with a specific design in the FPGA fabric, which cannot be reused for the PYNQ workflow.

To enable the proper configuration of the CIPS and boot the board with the static region design described in subsection II-B, in this work, a new petalinux build is performed with the PYNQ-specific design. The static hardware design can be exported from Vivado and used for configuring a stock petalinux project, with rootfs location set to SD card. To prepare the files for booting the device, the following command is called after performing the build:

```
petalinux-package --boot --plm
--psmfw --uboot --dtb --format MCS
```

Files from the boot partition of the MLIR-AIR image are then substituted by the packaged file from the petalinux build. At the boot stage, the FPGA fabric's static region is programmed, and the dynamic region from any Classic SoC Boot project can be used for programming the device.

B. PYNQ Library Modifications

This section describes the necessary modifications of the PYNQ Python library for obtaining a functional workflow on Versal ACAP platforms. It references specific library definitions, classes, and methods from the PYNQ library, whose definitions and detailed descriptions can be found in [16].

Since the device is programmed using a `.pdi` file instead of a bitstream, modifications to the `Overlay`'s `download` method were needed. Considering the changes in the interface of the FPGA manager driver on Zynq and Versal, its operation is mainly the same.

For Zynq, the binary data is extracted from the bitstream and written to a `.bin` file. This file is copied to `/lib/firmware`. Partial bitstreams could also be downloaded, and in that case, a "1" is written to the `/sys/class/fpga_manager/fpga0/flags` driver file. The download of the design to the FPGA configuration manager is then initiated by writing the bin filename to `/sys/class/fpga_manager/fpga0/firmware`.

For Versal, the full bitstream download is impossible, so the partial bitstream is always configured, and there is no need for a write to the `flags` driver file. To download a `.pdi` file, it is copied to `/lib/firmware`, and its name is written to the `firmware` driver file. This is very similar to the previous architecture, so a `PdifileHandler` with similar behavior to `BitfileHandler` was added to a list of bitstream handlers in the `embedded_device.py` file of the PYNQ library.

For Zynq, after a download, the processing system is reconfigured to accommodate the AXI4 interface size of the design. This is done by accessing the register map of the processing system. Since the Classic SoC Boot has the bus widths determined in the static part of the design, the library was modified to stop the reconfiguration in order to prevent runtime errors.

For allocation of the contiguous memory arrays, the newer versions of PYNQ, including v2.7 used here, use the Xilinx Runtime (XRT) library. Since the Versal device

is not compatible with the XRT library installed on the built image, during this work it was not possible to get it running. Instead, the legacy Xlnk class from the previous versions of PYNQ was utilized. It uses the `/dev/xlnk` device as an interface to the kernel. To enable the `/dev/xlnk` on the design, a device tree overlay was made with the following entry: `xlnk { compatible = "xlnx,xlnk-1.0"; };`, and the overlay target path "/". After loading the device tree, the Xlnk library works in the same way as legacy PYNQ versions. In order to maintain the same API as the v2.7, a wrapper was made that enables calls of the `pynq.allocate` method.

The Vivado tool generates hardware handoffs for both the static and the dynamic device images. Since all the memory-mapped peripherals of interest are placed in the dynamic portion of the design, the dynamic device image hardware handoff should be used alongside the `.pdi`. Since no major changes in the `.hwh` formatting were made for the new architecture, the existing parser operates normally, without modifications. The underlying method which is used for accessing the peripheral memory is to call `mmap` of the `/dev/mem`. Fundamentally, this means that the PYNQ library is able to interpret the entire physical memory of the processing system as one large array that is indexed by the memory address. Offsets that are parsed from the `.hwh` are used for selecting the absolute address. This proved to be a very effective way of abstracting the hardware access, and since it is device-agnostic, there were no changes required.

IV. Results and Discussion

The PYNQ environment functionality was successfully ported to the new Versal ACAP devices. During the course of our work, the VCK190 development board from AMD was used for development and testing, but the procedure should be easily replicated for other Versal devices. For the entire work, Vivado 2021.2 and petalinux 2021.2 were used, since they were compatible with the MLIR-AIR compilation flow. The process should be easily transferrable to newer versions.

In terms of the RTL design, a slightly modified workflow in Vivado is needed. The user should implement a new block design container with the specified interface, and configure it for dynamic implementation. At the time of the writing of this paper, this process still involves some manual configurations in Vivado, so further work on `.tcl` scripts should be done.

A screenshot of Jupyter Lab executing a test of the design from Fig. 4 on the VCK190 board is shown in Fig. 5.

V. Conclusion

This paper introduces a method for using a powerful PYNQ environment on Versal ACAP devices. To the best of the author's knowledge, this is the first time that the functionality of the PYNQ library has been adapted to work on a Versal platform. This work enables a productive workflow that simplifies the design, testing, and verification of FPGA IP cores that are particularly built and optimized for Versal. As Versal devices introduce many novelties when compared to the previous families, this work gives the user the ability to access those powerful new features using a simple interface.

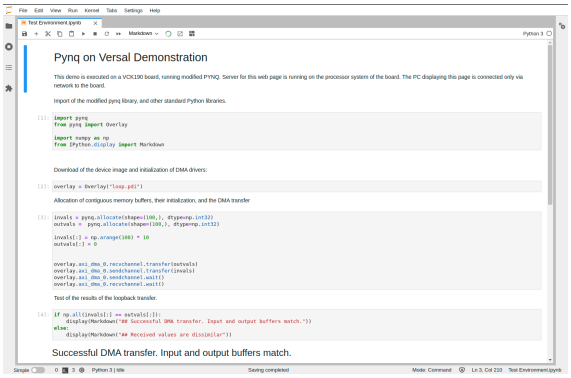

Figure 5. Screenshot of the Jupyter Lab window, with a demo of the PYNQ functionality on Versal

As a future work, the final goal of this project is to implement a complete Vivado framework that enables the implementation of the same block designs for both Versal and Zynq platforms, with identical PYNQ code used for testing.

References

[1] K. Vissers, "Keynote 2: Versal: The new xilinx adaptive compute acceleration platforms (ACAP)," in *2018 IEEE/ACM 8th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*. IEEE, Nov. 2018.

[2] (2023) Versal ACAP prime series. [Online]. Available: https://www.xilinx.com/products/silicon-devices/acap/versal-prime.html

[3] "Versal: The first adaptive compute acceleration platform (ACAP)," Xilinx Inc., White Paper WP505, Sep. 2020.

[4] "AI engines and their applications," Xilinx Inc., White Paper WP506, Dec. 2022.

[5] (2023) Python productivity for Zynq PYNQ. [Online]. Available: http://www.pynq.io/

[6] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy array: A structure for efficient numerical computation," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 22–30, mar 2011.

[7] G. Corradi, "The value of python productivity: Extreme edge analytics on xilinx zynq portfolio," Xilinx Inc., White Paper WP506, Jun. 2018.

[8] E. Wang, J. J. Davis, and P. Y. K. Cheung, "A PYNQ-based framework for rapid CNN prototyping," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, Apr. 2018.

[9] F. Kastner, B. JanBen, F. Kautz, M. Hubner, and G. Corradi, "Hardware/software codesign for convolutional neural networks exploiting dynamic partial reconfiguration on PYNQ," in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, May 2018.

[10] (2023) PYNQ support for versal (VCK190). [Online]. Available: https://discuss.pynq.io/t/pynq-support-for-versal-vck190/5470/2

[11] I. Swarbrick, D. Gaitonde, S. Ahmad, B. Jayadev, J. Cuppett, A. Morshed, B. Gaide, and Y. Arbel, "Versal network-on-chip (NoC)," in *2019 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, Aug. 2019.

[12] "Versal ACAP design guide," Xilinx Inc., User Guide UG1273, Jun. 2021.

[13] (2022) Vivado design tutorials: Classic soc boot. [Online]. Available: https://github.com/Xilinx/Vivado-Design-Tutorials/tree/2021.2/Device_Architecture_Tutorials/Versal/Boot_and_Config/Classic_SoC_Boot

[14] (2021) PYNQ SD card image. [Online]. Available: https://pynq.readthedocs.io/en/v2.7.0/pynq_sd_card.html

[15] (2021) MLIR-AIR platform. [Online]. Available: https://xilinx.github.io/mlir-air/

[16] (2021) PYNQ github repository, v2.7, library folder. [Online]. Available: https://github.com/Xilinx/PYNQ/tree/image_v2.7/pynq