Maximilian Eibl, Martin Gaedke. (Hrsg.): INFORMATIK 2017,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 481

A Hardware Accelerator Framework Approach for Dynamic Partial Reconfigurable Overlays on Xilinx PYNQ

Benedikt Janßen[1], Tim Wingender[2], Michael Hübner[3]

Abstract: Reconfigurable System-on-Chips (SoC) combine processor cores with Field-Programmable Gate Array (FPGA) fabric. Thereby, these systems enable to optimize the execution of application to some extend by hardware accelerators in the FPGA fabric. However, hardware accelerator development requires special skills from the developer since hardware development differs substantial from software development. Overlays offer a way to abstract the complexity of FPGA usage by predefined programmable hardware architectures. With Dynamic Partial Reconfiguration (DPR) it becomes possible to exchange parts of an overlay's architecture without affecting the operation of the remaining parts. In this article, we present a first version of our framework to ease the integration of hardware accelerators in Python via DPR overlays. The integration into Python is based on Xilinx PYNQ. The framework approach is based on our Python package 'pynqpartial'.

Keywords: Hardware Accelerator; FPGA; Python; PYNQ

1 Introduction

Application-specific hardware accelerators enable an optimized application execution. Besides the implementation into an application-specific integrated circuit (ASIC), Field-Programmable Gate Arrays (FPGAs) offer another more flexible way for the implementation. Their functionality can be defined by downloading configuration bitstreams in the field. The flexibility to program the hardware design in the field comes at the cost of an increased number of transistors and thus, in chip area and power consumption. However, it has been shown that FPGAs often offer a beneficial performance per watt ratio in comparison to pure processor and graphic processing unit (GPU) based designs [Le16][Nu16][AKJH16].

The higher optimized execution comes at the cost of an increased programming complexity. Within the FPGA research community, FPGA overlays are an active field of research to overcome this hurdle. Overlays abstract the complexity and challenges of hardware design by offering a predefined but programmable hardware architecture which is optimized for a class

[1] Ruhr-University Bochum, Embedded Systems for Information Technology, Universitätsstrasse 150, 44780 Bochum, Germany benedikt.janssen@ruhr-uni-bochum.de
[2] Ruhr-University Bochum, Embedded Systems for Information Technology, Universitätsstrasse 150, 44780 Bochum, Germany tim.wingender@ruhr-uni-bochum.de
[3] Ruhr-University Bochum, Embedded Systems for Information Technology, Universitätsstrasse 150, 44780 Bochum, Germany michael.huebner@ruhr-uni-bochum.de

doi:10.18420/in2017_44

of applications. Even though overlays are not optimized for a specific application, it has been shown that it is possible to maintain a high performance [CA13][JMF16][AKJH16][Pu14].

If an overlay targets an increasing bandwidth of functionality, its hardware resource requirements will increase too. Thereby, the resources available in the FPGA fabric are a limitation for such comprehensive overlays. Dynamic partial reconfiguration (DPR) offers a way out by enabling a temporarily shared usage of FPGA resources among different overlay processing functions. This sharing is done in a time-multiplexed manner. Thus, DPR enables a better resources utilization and resources efficiency [Ri16b][Ri16a][NL16]. For DPR a partial bitstream is loaded onto the FPGA, targeting only a certain part of its area. Within current FPGAs, this area need to be predefined during hardware design time and are called reconfiguration partition (RP).

The goal of our work is to enable a better integration of hardware accelerators into the software domain. To achieve this goal, we follow two approaches. First, to ease the hardware accelerator usage for software developers, and second, ease hardware accelerator interface creation for hardware developers. Therefore, we are developing a framework to support hardware developers to integrate their hardware into Python. Our work extends the Xilinx PYNQ project and targets the Xilinx Zynq-7000 System-on-Chip (SoC). The Zynq consists of an ARM A9 dual-core processor, denoted as processing system, and FPGA fabric, denoted as programmable logic.

The next Section 2 of this article presents the work related to our research. It is followed by a detailed explanation of our approach in Section 3 and a description of the implementation in Section 4. We evaluated the current version of our framework on the PYNQ-Z1 with multiple basic applications. The evaluation setup and results are presented in Section 5. Finally, Section 6 concludes the presented work and gives an outlook to future work.

2 Related Work

The Xilinx PYNQ open-source project was launched in 2016. The project goal is to enable the utilization of the Zynq's processing system and programmable logic through the productivity language Python [Xi16]. On the software side, PYNQ consists of a Python package, called 'pynq', which is shipped with a Linux image for the Xilinx Zynq. On the hardware side, the project develops an overlay that helps to evaluate the possibilities of the FPGA fabric from Python. By the time this article was written, the project supports the PYNQ-Z1 board as hardware platform. Subsection 3.1 analyses the PYNQ project more detailed.

Integrating hardware accelerators into the software domain is an ongoing research topic within the FPGA community. The basic communication mechanism between hardware and software in [Ka13] and [PK15] is memory-mapped register access. It is also the basis of hardware interface of the pynq package.

Xilinx SDSoC is a high-level synthesis (HLS) development environment for the Zynq-SoC. SDSoC enables a semi-automated implementation of hardware/software co-designs. For basic implementations, the developer does not need to be familiar with hardware development mechanism, e.g. hardware description languages. However, to guide SDSoC to an optimized implementation through directives, certain hardware properties and behavior need to be understood. The hardware/software co-design implemented by SDSoC is either based on Linux image which contains the application or the application is compiled into a bare-metal executable. For both cases, wrapper functions for the hardware access are inserted. In [KG16], the authors describe their work to enable DPR designs within SDSoC. They used the Tcl script interface of SDSoC to invoke the necessary commands. This way, the FPGA fabric can be utilized through comprehensive application-specific hardware accelerators. For their implementation, only minor hardware development knowledge is necessary.

FUSE targets to abstract the architecture of reconfigurable systems from the software developer [IS11]. FUSE is a front-end user framework to manage hardware accelerators. Software tasks represent the loaded accelerators to the software developers and enable a POSIX thread based API. This way, software developers do not need to have knowledge about the HW accelerator. Instead, it can be used through multithreaded programming models.

'hthreads' is a computational architecture created by Peck et al. [Pe06]. It has real-time and multithreading support, as well as POSIX-like threads. It is based on the concept of hybrid threads. The implementation is based on the hthreads microkernel. It is real-time capable, with support for multithreading and POSIX-like threads. The application source code is translated into a 'hardware intermediate form', which can be translated into VHDL.

ReconOS is a Linux-based operating system which provides libraries and drivers for hardware interaction [Ag14]. Moreover, its authors define a programming model and a dynamically reconfigurable architecture. The architecture developer is supported by a system builder tool, that integrates the system components. A distinguished feature of ReconOS is the migration of tasks from software to HW and vice versa.

In [AP16], the authors of hthreads and ReconOS, Andrews, Platzner, et al., summarize the goal of their work and provide a retrospective, with an emphasis on the programming model. They conclude that high-level programming models are a key for the success of reconfigurable manycore systems. Moreover, they point out that their results show, that tradeoffs between portability, scalability and performance are necessary. These key-objectives need to be considered when enhancing the hardware and software architectures.

R3TOS is a real-time operating system for reconfigurable devices [It13]. It supports hardware and software tasks and provides an API for developers. The communication between the hardware and software is based on buffers in a shared memory.

RIFFA is an open-source reusable integration framework for FPGA accelerators. In [JFK12],

Jacobsen and Kastner present the second version, RIFFA 2.0, which supports C/C++, Java 1.4+ and Python 2.7+. In software, byte arrays are used as data interface, the control of the hardware is done over PCIe register accesses. The communication is based on PCIe and hardware accelerators are interfaced with a FIFO-based protocol.

Sheffield et al. present 'Three Fingered Jack' [SAK12], a vectorising compiler and HLS tool to map certain Python loops onto an FPGA. They use a Python decorator '@fpga', to mark functions that should be redirected to their toolchain. The toolchain creates processing elements (PEs) that are placed in a cluster to compute the loop in parallel.

In [LKM16], Logaras et al. present SysPy, a Python-based tool to describe, verify and start the design implementation. Besides Python it is possible to describe parts of the system with VHDL. For the verification, it is possible to use Python-based testbenches and generate waveform diagrams. The implementation is supported by the generation of scripts for synthesis and implementation. Thus, SysPy is a Python-based hardware design tool for hardware developers than an application acceleration tool for software developers.

3 Framework Approach

Our framework approach utilizes the pynq package from the PYNQ project, as shown in Fig. 1. We chose this approach to build on top of an active project with a growing community. The framework utilizes the programmable logic management capabilities of the pynq package. The following Subsection 3.1 describes the interplay between our framework and the PYNQ components. The usage of DPR hardware accelerators through our framework is based on a DPR overlay concept that is described in Subsection 3.2.

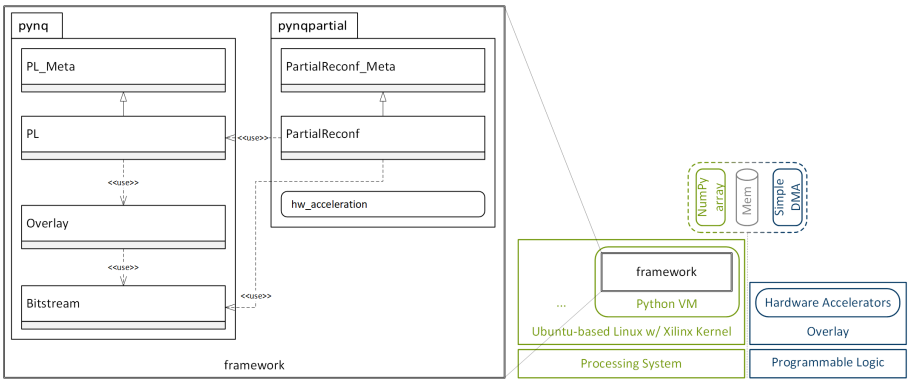

Fig. 1: Overview about the framework structure and its integration into PYNQ.

3.1 PYNQ Project Evaluation

The PYNQ project defines the web application Jupyter Notebook as its standard Python programming interface. Moreover, it uses the concept of overlays to utilize the programmable logic. It provides two classes for the developer to interact with the programmable logic itself, the 'Bitstream' and the 'Overlay' class. Moreover, it contains several classes to interact with hardware modules inside the programmable logic. The classes wrap the hardware by providing an API. It is also possible to wrap the FPGA configuration bitstream inside a Python package to enable an automated installation on other PYNQ systems via the Python package manager 'pip' [Ta16].

The programmable logic management is implemented within the 'PL' class, that inherits from the 'PL_Meta' metaclass. To avoid conflicts in the usage of the programmable logic between different classes, the PL class uses a client-server-approach based on the Python multiprocessing package to enable a central management. For our framework, we utilize the PL class to download bitstreams onto the programmable logic. This functionality is implemented within the 'download' method of the Bitstream class. The download is done through a driver provided by Xilinx and integrated into the PYNQ Linux image.

Moreover, the PYNQ Linux image includes another Xilinx driver that enables the allocation of physically contiguous memory. Physically contiguous memory is beneficial when working with hardware accelerators to enable simple direct memory access (DMA) from hardware, instead of complex scatter-gather DMA. The DMA class, included in the pynq package, provides a method to allocate physical contiguous memory from Python through the C Foreign Function Interface (CFFI). The translation of virtual memory from within Linux to physical addresses is implemented inside the 'MMIO' class that is based on the mmap system call. We utilize the MMIO class to access registers of the evaluation hardware.

3.2 Dynamic Partial Reconfigurable Overlay Concept

Our DPR overlay concept adopts and integrates into Python's package concept. As described in Section 3.1, overlay bitstreams can be wrapped in Python packages. We extend this relation, to relate the package itself to the static part of a DPR overlay. The classes, contained in package module, implement the methods to communicate with the hardware accelerators. A specific class is related to a specific hardware accelerator, and thus to a specific set of partial bitstreams. We call those classes hybrid classes, as they incorporate hardware and software. The concept is depicted in Fig. 2.

When an instance of a hybrid class is created, the object exists in the memory of the processing system, as well as in the programmable logic. Therefore, when the instance is deleted, the memory, as well as the programmable logic area needs to be freed.

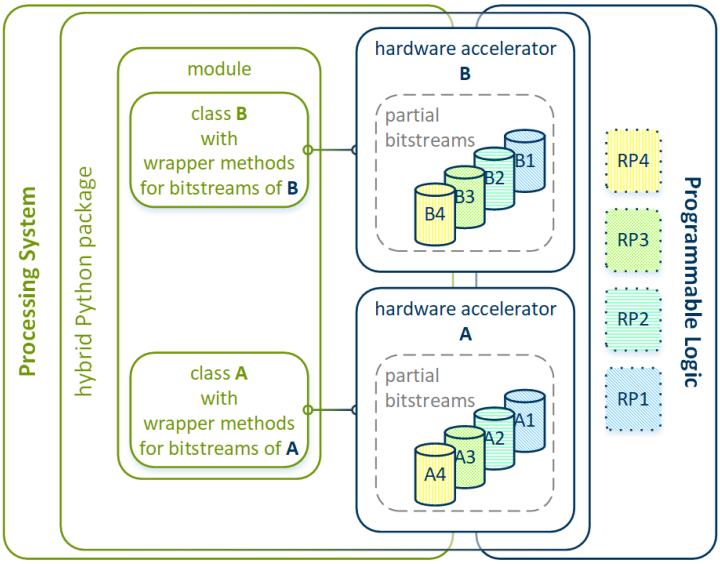

Fig. 2: Overview about the hybrid Python packages concept for DPR overlays.

4 Framework Implementation

To implement our concept, the framework approach consists of the Python package 'pynqpartial' and a hardware design template. Moreover, it defines a hardware/software interface, as introduced in Fig. 1. This part of the work is described in Subsection 4.1. For the communication between hardware and software, we use physical contiguous memory buffers. For the data type of the buffers CFFI supported data types can be used. This interfacing is described in Subsection 4.2.

4.1 Python Package

On the software side, our framework depends on our Python package pynqpartial []. For this work, we extended the pynqpartial package by the decorator function 'hw_acceleration' code to serve as the framework basis.

The decorator '@hw_acceleration' creates a wrapper, which interacts with the hardware accelerator. The function to be accelerated is passed as an argument. The execution sequence is shown in Fig. 3.

Before the wrapper function is created, the compatibility between the hardware accelerator and the hardware platform is checked. This process is shown in Fig. 3a). Therefore, the overlay must include a hardware description file, which describes the compatibility of the hardware functions. For this hardware description file, we chose the JSON format.

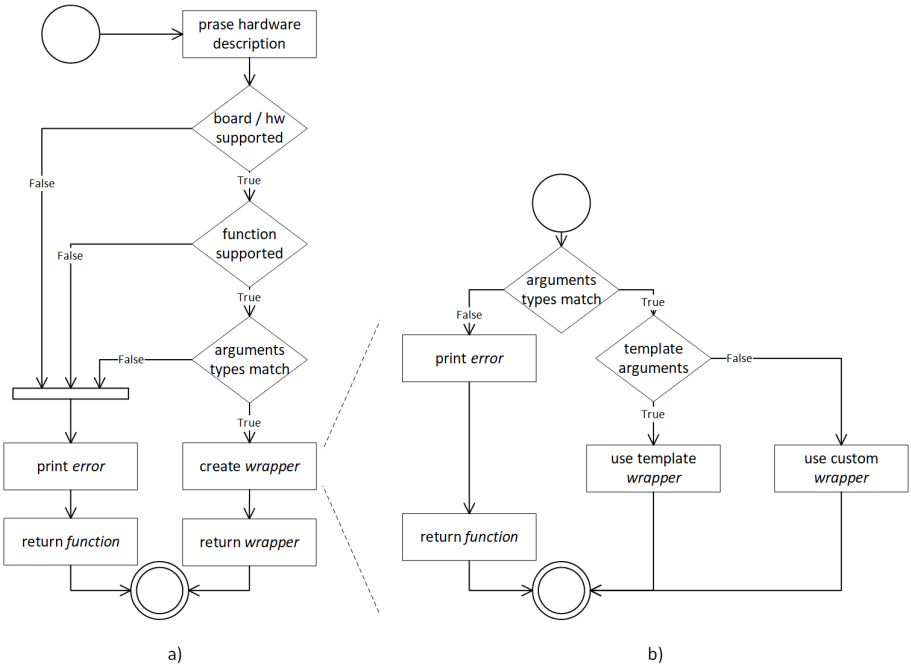

Fig. 3: Control flow sequence of wrapper creation (a) and wrapper execution (b).

Fig. 3b) shows the execution of the wrapper. Since Python is a dynamically typed language, we need to check the function arguments each time the function is called. With the first version of the framework, we support only keyword-only arguments. If the arguments match the requirements of our accelerator interface, a predefined wrapper function is returned. The following Subsection 4.2 describes the interfacing of the accelerators. The predefined wrapper is based on two instance of the DMA class to move data to and from the accelerator. For other accelerator interfaces a template function needs to be filled with the specific hardware calls by the hardware developer.

4.2 Accelerator Interface and Data Types

Currently, we support Xilinx's simple DMA cores as accelerator interface. Our wrapper is able to interface hardware accelerators with one or two input arrays and one output array. Any other hardware interface requires an adaptation of the hardware wrapper code. Since the DMA cores provide an AXI Stream interface, hardware accelerators should be compatible with this protocol. The input and output buffers need to be created manually by the application. With this first framework version, we evaluate NumPy arrays as interface for the hardware accelerators. As mentioned earlier currently up to two input buffers and

one output buffer are supported. The management is not optimized and involves several steps. The basic sequence for buffer allocation is listed below.

1. Create NumPy arrays for input and output data
2. Convert physical contiguous memory buffers to NumPy arrays
3. Copy data from NumPy input array to physical contiguous memory input array
4. Do hardware processing
5. Copy data from physical contiguous memory output array to NumPy output array

This flow has some potential for optimization, since (1) we currently need two NumPy arrays, for input and output data, and (2) the data needs to be copied from the NumPy arrays to the buffers. Aspect (1) depends on the application, for sequential memory accesses a single buffer can be sufficient. However, when dealing with random memory access patterns, it is likely that valid data will be overwritten. Aspect (2) depends on the memory allocation, NumPy arrays do not use physical contiguous memory, but virtual one. Therefore, we plan to introduce a data type which utilizes physical contiguous memory as data memory for NumPy arrays. This way, the software interface is a NumPy array and the hardware interface is a physical contiguous memory buffer.

5 Framework Evaluation

The focus of the evaluation lies on the decorator based hardware wrapper functionality. A data copy application, as well as a threshold filter have been used for the evaluation. The implementation is based on Xilinx Vivado HLS, its structure is depicted in Fig. 4.

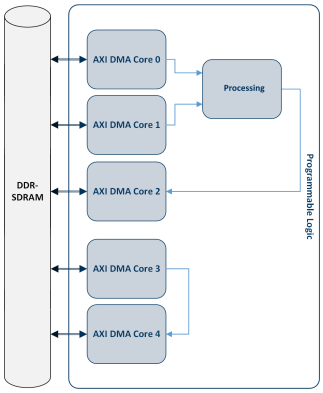

Fig. 4: Structure of the evaluation design; vector addition application uses three DMA channels, data copy application uses one per direction which are connected directly.

Fig. 5 shows the data copy application. We compare our hardware implementation against a for-loop, as well as NumPy's copy function. Since we are using NumPy.copy to copy the NumPy array data into the physical contiguous memory buffers, the hardware accelerated application is always slower. The execution time is not increasing significantly with an increase in data to be transfered. Therefore, it can be concluded that the communication overhead for the hardware is the bottleneck.

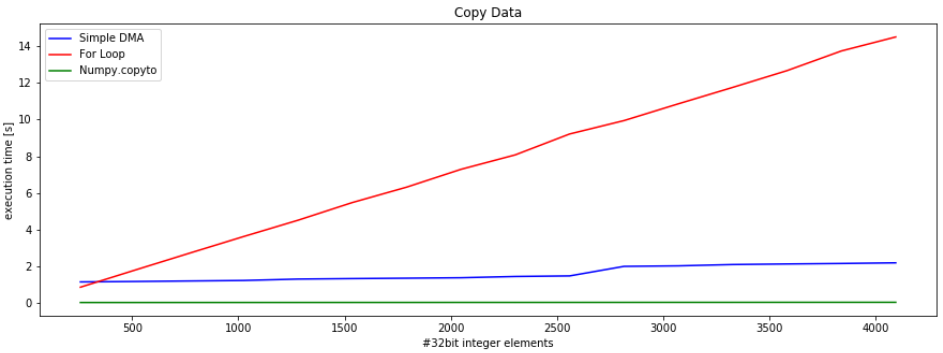

Fig. 5: Copy data application execution time over amount of data copied.

In the second evaluation benchmark we filtered four images with a threshold filter. The results are shown in Fig. 6. The hardware filter is about 56 times fast for processing the smallest image. For the larger images the speedup factor is around 98. The difference can be based on the communication overhead for configuring the DMA cores. The Python implementation is listed below. 'np_img' is a NumPy array that stores the image data.

```
np_img_flat = np_img.flatten()
for i in range(num_elmnt):
        if np_img_flat[i] >= thresh_values[0]:
                dstBuf_sw[i] = thresh_values[1]
        else:
                dstBuf_sw[i] = np_img_flat[i]
```

6 Conclusion and Outlook

In this article, we present our framework approach for DPR overlays on Xilinx PYNQ. We extended the pynqpartial package with function wrapper code to interact with hardware accelerators interfaces. Currently, only simple DMA cores are supported as hardware interface without manual adaption. The basic handling of the FPGA fabric is done via the 'Overlay' and the 'Bitstream' class of the pynq package. To use our framework, minor modifications to the pynq package are necessary.

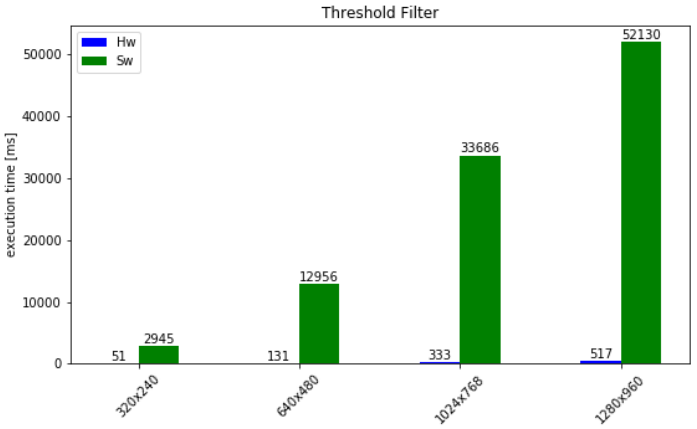

Fig. 6: Threshold filter execution time over image size.

In comparison with the related work, we focus on the interface between software and hardware developers. Therefore, unlike [SAK12], we do not target a just-in-time acceleration of software, but a library based acceleration. Moreover, we do not target an eased implementation of hardware accelerators, as for instance [LKM16], but a simplification of interface building for hardware developers. In comparison to the approaches taken by the authors of FUSE [IS11], hthreads [Pe06], ReconOS [Ag14] and R3TOS [It13], our hardware management is done at a higher level within Python. Therefore, it can be ported to other Linux-based systems on Zynq that support Python.

With this first framework version, we evaluated our approach. Up until now we do not have a comprehensive performance analysis, since our goal was to evaluation the functionality of our approach. Therefore, we will proceed with an analysis of the performance and bottlenecks, as well as requirements for more complex applications. For the future we plan to evaluate our framework with a data processing application from the process industry and extend it further based on the requirements of this application. Another aspect is the evaluation of a specific hardware type to avoid unnecessary memory allocation.

Acknowledgment

This work was done under the support of BMWi project MiMEP (03ET1314A).

References

[Ag14] Agne, A.; Happe, M.; Keller, A.; Lübbers, E.; Plattner, B.; Platzner, M.; Plessl, C.: ReconOS: An Operating System Approach for Reconfigurable Computing. IEEE Micro, 34(1):60–71, Jan 2014.

[AKJH16] Al Kadi, Muhammed; Janssen, Benedikt; Huebner, Michael: FGPU: An SIMT-Architecture for FPGAs. In: Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. FPGA '16, ACM, New York, NY, USA, pp. 254–263, 2016.

[AP16] Andrews, D.; Platzner, M.: Programming models for reconfigurable manycore systems. In: 2016 11th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC). pp. 1–8, June 2016.

[CA13] Capalija, D.; Abdelrahman, T. S.: A high-performance overlay architecture for pipelined execution of data flow graphs. In: 2013 23rd International Conference on Field programmable Logic and Applications. pp. 1–8, Sept 2013.

[IS11] Ismail, A.; Shannon, L.: FUSE: Front-End User Framework for O/S Abstraction of Hardware Accelerators. In: 2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines. pp. 170–177, May 2011.

[It13] Iturbe, X.; Benkrid, K.; Hong, C.; Ebrahim, A.; Torrego, R.; Martinez, I.; Arslan, T.; Perez, J.: R3TOS: A Novel Reliable Reconfigurable Real-Time Operating System for Highly Adaptive, Efficient, and Dependable Computing on FPGAs. IEEE Transactions on Computers, 62(8):1542–1556, Aug 2013.

[JFK12] Jacobsen, M.; Freund, Y.; Kastner, R.: RIFFA: A Reusable Integration Framework for FPGA Accelerators. In: 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines. pp. 216–219, April 2012.

[JMF16] Jain, A. K.; Maskell, D. L.; Fahmy, S. A.: Are Coarse-Grained Overlays Ready for General Purpose Application Acceleration on FPGAs? In: 2016 IEEE 14th Intl Conf on Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress(DASC/PiCom/DataCom/CyberSciTech). pp. 586–593, Aug 2016.

[Ka13] Kadi, M. A.; Rudolph, P.; Gohringer, D.; Hubner, M.: Dynamic and partial reconfiguration of Zynq 7000 under Linux. In: 2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig). pp. 1–5, Dec 2013.

[KG16] Kalb, T.; Göhringer, D.: Enabling dynamic and partial reconfiguration in Xilinx SDSoC. In: 2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig). pp. 1–7, Nov 2016.

[Le16] Lee, M.; Hwang, K.; Park, J.; Choi, S.; Shin, S.; Sung, W.: FPGA-Based Low-Power Speech Recognition with Recurrent Neural Networks. In: 2016 IEEE International Workshop on Signal Processing Systems (SiPS). pp. 230–235, Oct 2016.

[LKM16] Logaras, E.; Koutsouradis, E.; Manolakos, E. S.: Python facilitates the rapid prototyping and hw/sw verification of processor centric SoCs for FPGAs. In: 2016 IEEE International Symposium on Circuits and Systems (ISCAS). pp. 1214–1217, May 2016.

[NL16] Nafkha, A.; Louet, Y.: Accurate measurement of power consumption overhead during FPGA dynamic partial reconfiguration. In: 2016 International Symposium on Wireless Communication Systems (ISWCS). pp. 586–591, Sept 2016.

[Nu16] Nurvitadhi, E.; Sim, Jaewoong; Sheffield, D.; Mishra, A.; Krishnan, S.; Marr, D.: Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC. In: 2016 26th International Conference on Field Programmable Logic and Applications (FPL). pp. 1–4, Aug 2016.

[Pe06] Peck, W.; Anderson, E.; Agron, J.; Stevens, J.; Baijot, F.; Andrews, D.: Hthreads: A Computational Model for Reconfigurable Devices. In: 2006 International Conference on Field Programmable Logic and Applications. pp. 1–4, Aug 2006.

[PK15] Prince, A. A.; Kartha, V.: A framework for remote and adaptive partial reconfiguration of SoC based data acquisition systems under Linux. In: 2015 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC). pp. 1–5, June 2015.

[Pu14] Putnam, Andrew; Caulfield, Adrian M.; Chung, Eric S.; Chiou, Derek; Constantinides, Kypros; Demme, John; Esmaeilzadeh, Hadi; Fowers, Jeremy; Gopal, Gopi Prashanth; Gray, Jan; Haselman, Michael; Hauck, Scott; Heil, Stephen; Hormati, Amir; Kim, Joo-Young; Lanka, Sitaram; Larus, James; Peterson, Eric; Pope, Simon; Smith, Aaron; Thong, Jason; Xiao, Phillip Yi; Burger, Doug: A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In: Proceeding of the 41st Annual International Symposium on Computer Architecuture. ISCA '14, IEEE Press, Piscataway, NJ, USA, pp. 13–24, 2014.

[Ri16a] Rihani, M. A.; Nouvel, F.; Prévotet, J. C.; Mroue, M.; Lorandel, J.; Mohanna, Y.: Dynamic and partial reconfiguration power consumption runtime measurements analysis for ZYNQ SoC devices. In: 2016 International Symposium on Wireless Communication Systems (ISWCS). pp. 592–596, Sept 2016.

[Ri16b] Rihani, M. A.; Prévotet, J. C.; Nouvel, F.; Mroue, M.; Mohanna, Y.: ARM-FPGA based platform for automated adaptive wireless communication systems using partial reconfiguration technique. In: 2016 Conference on Design and Architectures for Signal and Image Processing (DASIP). pp. 113–120, Oct 2016.

[SAK12] Sheffield, D.; Anderson, M.; Keutzer, K.: Automatic generation of application-specific accelerators for FPGAs from python loop nests. In: 22nd International Conference on Field Programmable Logic and Applications (FPL). pp. 567–570, Aug 2012.

[Ta16] Taylor, A.: Adam Taylor's MicroZed Chronicles Part 156: Pynq Hardware Overlays. 2016. https://forums.xilinx.com/t5/Xcell-Daily-Blog/A dam-Taylor-s-MicroZed-Chronicles-Part-156-Pynq-Hardware/ba-p/732835, [Online; accessed 2017-04-03].

[Xi16] Xilinx, Inc.: PYNQ: PYTHON PRODUCTIVITY FOR ZYNQ. 2016. https://www.pynq.io, [Online; accessed 2017-04-03].