UltraScale+ SpinalHDL Wrapper: Streamlining Ideas to Bitstream on UltraScale+ platforms.

Denis Hoornaert
Technical University of Munich
Munich, Germany
denis.hoornaert@tum.de

Giulio Corradi
Advanced Micro Devices
Munich, Germany
giulio.corradi@amd.com

Renato Mancuso
Boston University
Boston, Massachusetts, USA
rmancuso@bu.edu

Marco Caccamo
Technical University of Munich
Munich, Germany
marco.caccamo@tum.de

***Abstract*—In an embedded computing landscape that inexorably leans into heterogeneity, System-on-Chips (SoCs) featuring tightly integrated Field Programmable Gate Arrays (FPGA) are bound to proliferate. In particular, such architectures' high degree of flexibility and control caters well to the real-time community. Despite the appeal, real-time research exploiting HW/SW co-design on such architectures has remained tepid. While the usual suspects, such as the complexity of Hardware Description Languages, can be blamed, recent advancements in tooling (*e.g.*, languages, frameworks) have proven efficient in easing the design of FPGA-located accelerators. However, in the context of SoC with FPGA platforms, these solutions fall short of addressing the next hurdle: integrating the custom accelerators with the rest of the SoC, which requires the tedious implementation of various supporting software resources.**

This article presents the first iteration of the UltraScale+ SpinalHDL Wrapper; a SpinalHDL library dedicated to supporting HW/SW co-design on SoC with FPGA platforms. The support ranges from assisting during the design of accelerators to automatically inferring and generating ready-to-use software support, such as Linux Kernel modules and Vivado deployment scripts.

***Index Terms*—FPGA, UltraScale+, Hardware/Software co-design, Hardware Construct Languages**

I. Introduction

Modern System-on-Chip (SoC) for embedded systems are becoming more performant as the functional requirements have prompted a surge in computational demand. Combined with use case-specific Size, Weight, and Power (SWaP) constraints, this has pushed SoC architectures to become increasingly heterogeneous by integrating highly specialized accelerators. Nowadays, platforms featuring on-chip specialized units (*e.g.*, GPUs, TPUs) are ubiquitous. Particularly, SoC architectures featuring tightly integrated Field Programmable Gate Arrays (FPGAs) have attracted attention due to their inherent high degree of programmability and on-the-fly reprogrammability. The overall appeal of this SoC architecture is confirmed by the emergence of many platforms [1]–[4] among which the AMD-Xilinx UltraScale+ model [5] has garnered the most attention. This class of platforms is also referred to as *PS-PL* to denote the combination of a Processing System (PS) (*i.e.*, CPU cores and memory) and a Programmable Logic (PL) (*i.e.*, FPGA). This terminology is used in this article.

Since 2016, the opportunities offered by HW/SW co-design on UltraScale+ platforms have caught the attention of the embedded real-time community. Much research has focused on implementing PL-located accelerators, with application domains ranging from image processing [6] to artificial intelligence [7]. Several research groups have also investigated the impact of inter-accelerator bus activity on the latter's performance [8], [9] and proposed mitigation policies [10], [11]. Frameworks [12], [13] tying together the FPGA's support for dynamic and partial reprogramming with scheduling models have been proposed. Finally, a series of papers explored the idea of using the PL as part of the SoC's memory system, enabling memory requests manipulation [14], [15] and traffic regulation [16], [17]. However, despite the many potentials, these research threads have remained tepid within the embedded real-time community, as illustrated in Fig. 1.

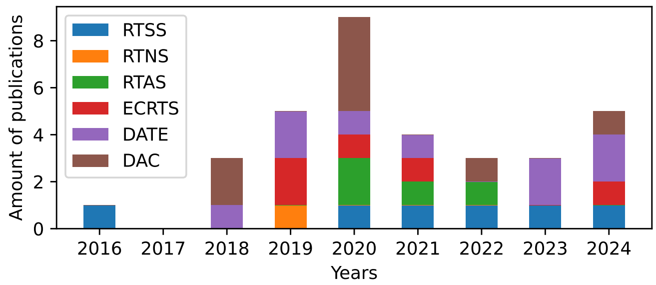


Fig. 1: Published paper since 2016 employing HW/SW co-design on UltraScale+ in real-time oriented conferences.

This trend can be in part associated with the notorious difficulty of designing accelerators for FPGAs compared to programming for other Processing Elements (PEs, *e.g.*, CPUs, GPUs). The commonly agreed-upon culprits are the *traditional* Hardware Description Languages (HDLs, *e.g.*, VHDL, Verilog). Aspects such as their verbosity and confusing specification model are often considered key factors slowing down productivity and learning rate. In that regard, considerable efforts from the research community and the industry have led to the development of advanced approaches and tooling to speed up hardware design time. Notably, High-level Synthesis (HLS) and Hardware Construct Languages (HCL) have addressed these issues by using higher abstraction languages to design and generate digital circuits.

Unfortunately, however, in the context of PS-PL platforms,

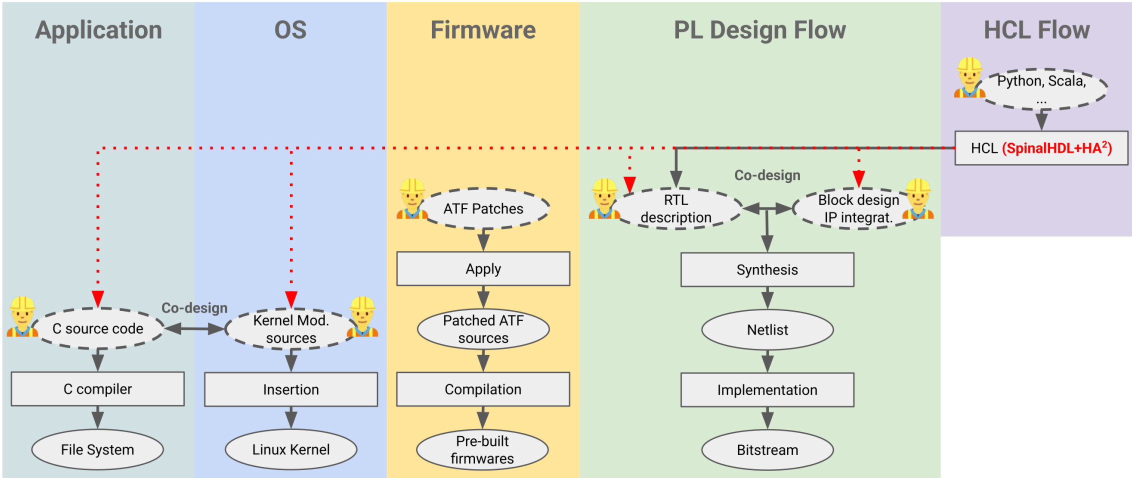


Fig. 2: Overview of the co-design flow for UltraScale+ platforms. Each layer of the system involves manual work requiring specialized expertise (dashed ellipse). The *"PL Design Flow"* can be extended by a HLS or HCL. Our SpinalHDL+USW (dotted red arrows) only requires human intervention for specifics about the hardware, firmware, and final user application.

the aforementioned tools do not address the next productivity hurdle: *hardware/software interaction*. Existing tools [18], [19] focus solely on the accelerator's logic while integration tools [20] consider the PL side in isolation, leaving designers with the tedious—yet delicate—task of integrating the FPGA accelerator with the rest of the system by implementing the necessary software drivers. Nonetheless, proper integration on such a platform is challenging because it may require firmware and Operating System (OS) expertise.

As the authors found out, after several years of experience with the UltraScale+ platform, most of the productivity hurdles often correspond to repetitive implementation tasks. As such, instead of curating their experience as a set of guidelines and documents, the authors propose the UltraScale+ SpinalHDL Wrapper (USW), an open-source [21] library extending a HCL (*i.e.*, SpinalHDL) that aims to generate ready-to-use support to ease HW/SW co-design on AMD-Xilinx UltraScale+ platforms. Such support includes (1) pre-defined specialized hardware constructs for several UltraScale+ boards (*e.g.*, Kria KV260 and Kria KR260), (2) the generation of Linux kernel modules, and (3) the generation of AMD-Xilinx Vivado TCL scripts to enable "one command line" deployment. Through this, USW aims to lower the entry barrier of HW/SW co-designing on UltraScale+ platforms.

II. Ancillary Concepts

A. FPGA and Hardware Description Language

A Field Programmable Gate Array (FPGA) is a type of re-programmable PE capable of emulating *virtually* any specialized digital circuits. FPGAs sit at the cross-road between specialized hardware accelerators and general purpose (*e.g.*, ISA-based) PEs. Like the latter, FPGAs can implement workload-tailored data manipulation and display aggressive parallelism (*e.g.*, via pipelining) while, like general-purpose PEs, they can be dynamically re-programmed on demand.

The design of digital circuits in FPGA is done via HDLs (*e.g.*, Verilog, VHDL), which uses the Register Transfer Level (RTL) abstraction. As the names suggest, with these languages, one *describes* the data flow from registers to registers, wires to wires, and vice versa. An Electronic Design Automation tool (*e.g.*, Vivado) is employed to synthesize the RTL description into a logically equivalent and target-specific *bitstream* that can be flashed onto the FPGA. This process, illustrated in Fig. 2 (green-shaded box), is performed through a sequence of steps referred to as the *PL design flow*. Essentially, once the HDL description is created, it must be connected and co-designed with—potentially vendor-locked—third-party IPs. The output design is *synthesized* into an intermediate representation called a *netlist* on which a series of optimizations are applied. The *implementation* step yields the bitstream.

Due to the tediousness of designing hardware accelerators with traditional HDLs, many projects and industrial products have sought to lift the level of abstraction. Existing solutions rely on CPU programming languages to generate HDLs, sitting atop the usual design flow as shown in Fig. 2. Two notable approaches exist. (1) ***HLS*** [22], [23] aim at transpiling C code into HDL such as Verilog to take advantage of the existing code base and enable a fast *time-to-bitstream* for non-HDL experts. However, the procedural-to-RTL transformation is not straightforward and requires designers to expertly guide the tools via `C` pragmas. (2) ***HCL*** such as Chisel [19] and SpinalHDL [18] take a different route. They posit that the limited code re-utilization and associated language features of HDLs are the main productivity hurdle, not the RTL abstraction. Hence, these HCLs still use the RTL abstraction but embed it within high-level programming languages such as Scala. The latter acts as both a framework and a pre-processing language, offering object-oriented and functional programming features, as well as expressiveness to elaborate and test the designs. Because the final hardware description

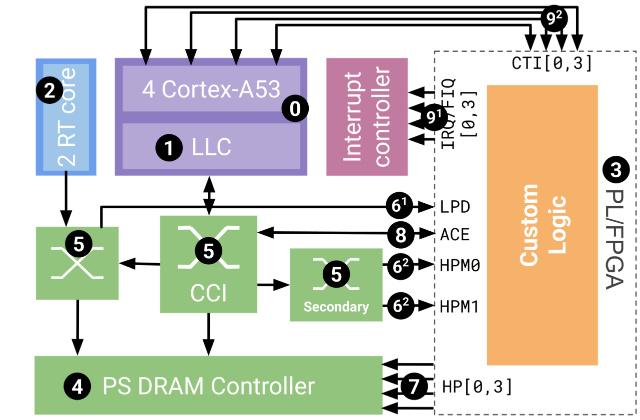


Fig. 3: Overview of a UltraScale+ SoC where only relevant components are depicted. Components are marked by ⓧ.

uses the RTL abstraction, unlike HLSs, the final logic corresponds 1-to-1 to the desired semantics, avoiding any *"black-magic"* [18] transformation.

B. PS-PL platforms

As depicted in Fig. 3, the UltraScale+ multi-processor SoC manufactured by AMD-Xilinx features four ARM Cortex-A53 CPU cores ⓿ equipped with a 1 MB last-level cache ❶, a pair of real-time ARM CPU cores ❷, and an FPGA ❸ (*i.e.*, the PL side). These elements and the DRAM controller ❹ are linked via a system of interconnects ❺ and dedicated signals.

A key aspect of co-designing PS software with PL accelerators is their interactions. On UltraScale+, PS-PL interaction is facilitated by a diverse set of interfaces, including: (1) seven unidirectional AXI4 interfaces: (a) three of whom are memory mapped PS-to-PL interfaces used by the PS to fetch data from the PL ❻ and (b) four PL-to-PS interfaces that allow access to any on-chip memories and the main memory ❼. (2) One two-way cache coherent interface (ACE) that allows the PL to become a member of the SoC cache coherence domain ❽ (*i.e.*, snoop and be snooped by the CPU cores). (3) Several direct PS-to-PL and PL-to-PS interrupt ❾^1 and cross-trigger ❾^2 lines enable fast communications to/from the PS-side interrupt controller and CoreSight debugging infrastructure.

With an increased heterogeneity comes an increase in design complexity and scope. In fact, not only should designers take into account the PL, they must also prepare the PS and its various software requirements. In this widened co-design flow (shown in Fig. 2), each traditional layer of a system (*i.e.*, firmware, OS, and user/application) must be set up and tailored for communication with the PL-located accelerator(s). On UltraScale+ platforms, it comes as firmware patches (*e.g.*, to enable the PL-side ACE port) and OS kernel modules (*e.g.*, to map the various ports and interrupt lines).

III. Objectives and Overview

Unlike existing FPGA integration tools (*e.g.*, LiteX [20]), the proposed approach does not consider the PL side as an isolated block but instead focuses on its integration with the PS side. Based on the authors' experience, much of the software support required to smoothly interface software with PL-located accelerators can be automatically generated and, especially, inferred from the hardware module description. The proposed approach leans on this observation to ease HW/SW co-design on UltraScale+ platforms.

This article proposes to extend SpinalHDL [18] (an HCL) via a dedicated library called the UltraScale+ SpinalHDL Wrapper (USW). The library assists the hardware designer by providing pre-defined hardware constructs common to all UltraScale+ boards and implicitly generating ready-to-use software resources associated with them. As illustrated in Fig. 2 by the dashed red arrows, the software resources generation ranges from Vivado TCL script to kernel modules. Note that using USW does not lock the designer in, as the generated support can be used as is or as the starting point toward further (manual) customization.

This section presents some of the constructs and tools offered by USW. The code snippet displayed in Listing 1 exemplifies the use of USW; however, details regarding SpinalHDL syntax will not be presented due to space constraints. For further details, readers are invited to look at [18].

A. Top Module, Interfaces, and I/Os

The top module is designed such that it contains all PL's elements. As such, it mirrors all PS-PL interfaces and I/Os. The idea is that the SpinalHDL description is self-contained, and no subsequent manipulation of the outcome is needed.

This implies that most information must come or be derived from the SpinalHDL description. In the proposed library, information such as (1) the target board, (2) the target frequency, and (3) the desired interfaces and I/Os are specified directly in the top-module definition. The top module can be created via a Scala class that inherits a curated class describing the target board (*e.g.*, Kria KV260, Kria KR260, or ZCU102). For instance, the code snippet in Listing 1 implements a top module called `ConfigTestPort` on the Kria KV260 (see line 9). This parent class takes two construction parameters: a frequency and an I/O configuration.

The frequency parameter indicates the desired synthesis (line 10) but does not imply that the Verilog produced can operate at that frequency. The library provides an early indication of whether the target board can support the frequency, sets the closest if not, and considers it when generating the Vivado TCL script (see Sec. III-D). The I/O configuration indicates which interfaces and I/Os should be implemented (lines 11-13). The associated apertures become available in the description block by enabling specific options. In our example, asserting `withLPD_HPM0` tethers the design to the `LPD HPM0` PS-PL interface and allows access to the individual fields in the description via `io.lpd.hpm0` (❻^1 in Fig. 3).

B. Configuration Port

A configuration port is a key component of any accelerator on UltraScale+ as it allows the PS-located software layers to instrument the PL-located accelerators. Implementing an

```
1  package example
2
3  import spinal.core._
4  import spinal.lib._
5  import kv260._
6  import ultrascaleplus.scripts._
7  import ultrascaleplus.configport._
8
9  case class ConfigPortTest() extends KV260(
10   frequency = 100 MHz,
11   config    = new KV260Config(
12     withLPD_HPM0 = true, withIO_PMOD0 = true
13   )
14 ) {
15   val configPort = ConfigPort(io.lpd.hpm0,
    ↪   io.lpd.hpm0.getPartialName())
16
17   val clockCount = Reg(UInt(64 bits)) init(0)
18   configPort.read(clockCount,
    ↪   io.lpd.hpm0.apertures(0).base)
19
20   clockCount := clockCount+1
21
22   for (i <- 0 until io.pmod0.length)
23       io.pmod(i) := clockCount(63-i)
24
25   KernelModule.addIO(io.lpd.hpm0)
26   KernelModule.generate()
27   this.generate()
28 }
```

Listing 1: SpinalHDL code snippet implementing an AXI4 config. port using USW for the Kria KV260 (line 9). Configuration port is tethered to the `LPD HPM0` (line 15) PS-PL interfaces and enables access to a 64-bit wide counter (lines 17-18).

AXI4 compliant configuration port is tedious, but, luckily, SpinalHDL provides an easy-to-use generator [24]. However, the generator lacks a few features to ease the integration with the PS side.

To this end, our USW library provides a specialized version: the `ConfigPort` generator. The latter extends via inheritance the SpinalHDL's generator to provide ready-to-use `C` code generation support for the PS side. More precisely, after being instantiated (line 17), registers can be added to the configuration in read-only, write-only, and read-write modes. For example, in Listing 1 line 18, the `clockCount` register is added for read-only access at address `io.lpd.hpm0.apretures(0).base`. At SpinalHDL elaboration-time, the USW library produces boilerplate `C` code that defines a `C` struct with a field for each register added to the configuration port. Their placement reflects the address specified in the hardware description (Listing 1, line 18). When required, padding is automatically added to the struct. The generated `C` struct provides the typical code to memory-map (`mmap`) the associated PS-PL AXI4 interface and cast the returned pointer into the generated struct.

C. Kernel Module

Part of the challenge when co-designing an accelerator with Linux is to allow access to the PS-PL interfaces. In particular, allocating address ranges falling within the `FPD HPM[0,1]` apertures as cacheable regions directly entails the involved alteration of the Linux kernel or the use of a hypervisor as done in [15], [16]. An alternative is to create an *"insertable"* kernel module that creates `/dev` file system targets. When these `/dev` targets are mapped (via `mmap`), they return a pointer to the base address of the desired aperture.

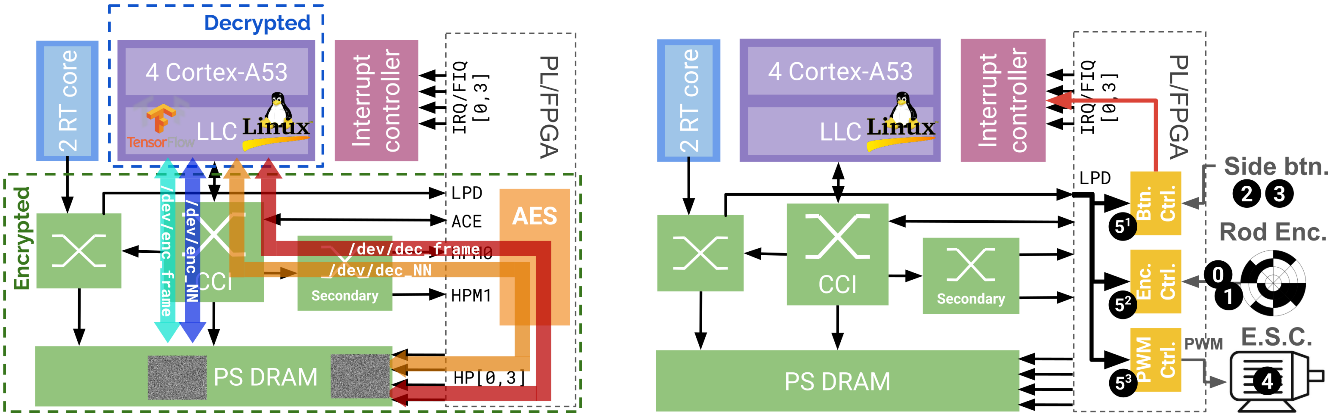
The library's kernel module generator leverages the latter option. As shown in Listing 1 line 28, it is as simple as invoking a singleton named `KernelModule` and calling its `addIO` method on the desired AXI4 port. In this case, the generated kernel module will create, after insertion, a `/dev/lpd_hpm0` file that, when mapped, allows I/O (*i.e.*, uncached) access to the region. Cacheable targets can be generated using the `add` method instead.

D. Vivado TCL Script

The generation of TCL scripts is supported and provided natively by Vivado and is the *de facto* preferred way to share and maintain versioning of Vivado projects. However, unless designers can expertly program in TCL from scratch, the design and all PS-PL I/O connections must be established manually over the GUI. This introduces a human-in-the-loop step that still requires some expertise with the tool, effectively keeping the entry barrier high and preventing fast prototyping. Moreover, any future changes to the design's PS-PL interfacing or in-use I/O entail manipulating and regenerating the TCL script using Vivado. Finally, Vivado projects' portability and deployment are challenging because the TCL scripts are version dependent, and licenses are available only for some versions.

Following the aforementioned philosophy, our USW library can generate a Vivado TCL script to easily share and deploy UltraScale+ designs at the elaboration of a SpinalHDL description (*i.e.*, from SpinalHDL to Verilog). The automated TCL script generation is made possible by the information provided in the description's top module. The produced and ready-to-use TCL scripts are placed in the `vivado/` folder and named after the design they implement—*e.g.*, `ConfigPortTest.tcl` for the design in Listing 1. A clear advantage of inferring the TCL script from the design is the augmented cross-version portability. Concretely, with USW, sharing the design sources is sufficient to allow any collaborator to seamlessly deploy and implement the design regardless of their installed Vivado version. We have tested that USW-generated TCL scripts work correctly for the 2024.2, 2024.1, 2022.2, and 2019.2 versions of Vivado and expect broad compatibility with other versions.

Like other USW-generated support, TCL scripts do not need to be regenerated every time. For instance, if no I/O nor PS-PL interface is altered, removed, or added, the TCL script can serve as an initial springboard toward further manual customization by the designer. Alternatively, the TCL script can be disregarded at the designer's discretion. Finally, note



(a) The AES accelerator receives cacheable transactions on the `FPD HPM0` port. The a-la PLIM module deciphers and ciphers data on reads and writes. The NN and frames remain encrypted in memory.

(b) The PL-located PWM controllers are configured by the CPUs via the `LPD HPM0` port to (1) emit one PWM signal aimed at controlling the pole's car and (2) maintain an estimation of the system's state.

Fig. 4: Abstract overview of the two example use-cases described in Sec. IV implemented on the AMD-Xilinx UltraScale+ platform using SpinalHDL and the USW library.

that any (*i.e.*, small or not) changes applied to a hardware design (*e.g.*, change of condition operator from `&` or `|`) obligatorily implies going down the PL design flow again. This is a lengthy process even for smaller projects, making the redeployment phase of the Vivado project via the USW-generated TCL script a hardly noticeable overhead.

E. SpinalHDL+USW to Bitstream

Once a SpinalHDL description exists, generating the bitstream is a matter of two command lines, as shown below. The resulting bitstream file is placed in the project's folder, and the Vivado project is created in the `vivado/` folder.

```
sbt "runMain example.ConfigPortTestVerilog"
vivado -source vivado/ConfigPortTest.tcl
```

IV. Demonstrators

As mentioned in Section I, HW/SW co-design on PS-PL platforms (and the UltraScale+ platform in particular) have already been successfully deployed in several real-time contexts. Naturally, USW can be used (or has already been used) to replicate the prototypes presented in [10], [15]–[17]. For instance, readers can find SpinalHDL support to generate PLIM-like IPs[1] and a simplified implementation of the PLIM-style *bleacher*[2] from [15] in the example designs of USW.

The authors selected two divergent HW/SW co-designed projects to illustrate the PS-PL interfaces and I/O generation abilities. These examples are individually presented in this section and illustrated in Fig. 4.

[1]https://github.com/denishoornaert/ultrascale-spinal-wrapper/blob/master/hw/spinal/example/Plim.scala

[2]https://github.com/denishoornaert/ultrascale-spinal-wrapper/blob/master/hw/spinal/example/Bleacher.scala

A. Security: On-the-fly AES Encryption

This use case, whose rationale is exhaustively presented in [25], aims to enable on-demand access to confidential data while always keeping the latter encrypted in memory. Typically, the PL-located accelerators encrypt/decrypt confidential data block-by-block upon explicit instrumentation by the PS CPU cores. Instead, as illustrated in Fig. 4a, the design follows the PLIM [15] approach to allow greater flexibility and smoother integration with the PS-located software. This approach creates an alternative PL-traversing route to memory featuring an encryption engine that encrypts and decrypts individual transactions.

Concretely, the encryption engine is implemented as a *counter-mode* AES-128 accelerator [26] from scratch using SpinalHDL+USW for the Kria KV260. The tethering of the three PS-PL interfaces (`LPD HPM0`, `FPD HPM0`, and `FPD HP0`) with the top module is realized via the USW library and the generation of the kernel modules mapping the two primary ports is delegated to the USW library (see Sec. III-C). The `LPD HPM0` port is attached to a configuration port (see Sec. III-B) that stores crucial information (*e.g.*, encryption key) and USW generates a ready-to-use `C` struct for HW/SW interfacing.

On the software side, the confidential (*i.e.*, encrypted) workloads consist of (1) the weights of neural network detecting the presence of humans (implemented and run via TensorFlow Lite) and (2) encrypted video frames coming from a remote device (*e.g.*, over the internet). This data is accessible via four `/dev` targets as illustrated in Fig. 4a (see arrows); two of which are generated by the USW library. More accurately, the orange and red arrows represent the data path created by the `/dev` targets generated by USW. They allow the software direct communication with the PL and access to the decrypted version of neural network (NN) weights (*i.e.*, `/dev/dec_NN`) and the video frame (*i.e.*, `/dev/dec_frame`) respectively.

From a subjective point of view, contributors to the project concurred that the establishment of the `/dev` targets and `C` struct as a high-level and well-specified interface rendered the HW/SW co-designing more approachable. In particular, it enabled better cross-team communication. This observation underscores the usefulness of streamlining software support.

B. Robotic: Inverted Pendulum

This demonstrator showcases the ability of the library to support HW/SW co-designed robotic systems. In such systems, software-implemented controllers running on the PS interact with custom PL-located peripheral (*i.e.*, I/O) controllers to sense and act on their environment. These controllers can also feature high-frequency stream processing (*i.e.*, accelerated) logic.

We selected a custom inverted pendulum mechanical system as a use case. To achieve the goal of balancing and maintaining the free-flowing rod in an upward position, the cart, actuated by a motor, is controlled by an LQR+Simplex controller. The latter interacts with the PL to set its action (*i.e.*, motor RPM) and access an estimation of the system's state.

For this use case, the Kria KR260 board is used. As illustrated in Fig. 4b, the PL hosts an *inverted pendulum* [27] design consisting of three controllers: two encoders with state estimation logic (⓿ and ❶), two buttons indicating each end of the rail (❷ and ❸), and one PWM controller (❹). The encoder controllers are tasked to monitor *all* switching activity on the `PMOD0` associated I/O lines and process them to update state estimation. In our use case, one provides an estimate of the rod position as an angle w.r.t. to the starting point (*i.e.*, pointing downwards), and the second one provides an estimate of the cart's position as a distance w.r.t. the middle of the rails. All controllers can be instrumented and their state accessed by the PS software via a configuration port (see Sec.III-B; ❺) generated and connected to the `LPD HPM0` port by USW. The PS-located controller utilizes the associated `C` struct and a kernel module (see Sec. III-C) generated by our USW framework.

V. Discussion

A. Scalability and Portability

The choice of the UltraScale+ platforms as a focus point is logical considering its widespread availability and omnipresence in the few real-time research using PS-PL platform for HW/SW co-design. However, the USW can be extended to support *virtually any* platform. This is greatly facilitated by the UltraScale+ SoC implementation of standard open-specification protocols (*e.g.*, AXI) as PS-PL interfaces. The combined flexibility and agile approach of USW and SpinalHDL ease the introduction of new interfaces and protocols.

Naturally, expanding USW support to include closely aligned SoCs such as AMD-Xilinx' Zynq and Versal are expected to be easier as they share a common interface naming convention and synthesis backend (*i.e.*, Vivado) with the UltraScale+ platform. The amount of effort required to support other platforms will vary depending on a few factors. For instance, the Enzian [3] platform would benefit from the same backend support as the PL is AMD-Xilinx based. However, some work will be required to add the bus protocol specific to their platform. On the other hand, platforms like [1], [2] rely on different backends, meaning that the TCL script generation may have to be revamped. This could be addressed by raising the abstraction level to provide a common interface and automatic backend selection. Interface-wise, expanding support is easier as the interface protocols used (*e.g.*, AXI, TileLink) are already supported by SpinalHDL.

B. PS-PL Interfaces, Interactions, and Timings

Many related research works cited in this paper already provide timing measurements for accessing PL and PS elements that traverse their shared interfaces. In particular, [28] provides an exhaustive analysis and result set for PL-to-PS communications. It shows that the PS and PL can access the main memory with up to $4.8\ GBps$ of throughput. In [15], the authors report that accessing PL-located memory blocks can be done at $\pm 800\ MBps$. These results provide insight into the level of performance one can expect when employing *"simple"* direct access means like the memory region mapping supported by USW (see Sec. III-C).

However, in the future, USW aims to enable the generation of more sophisticated means of interaction between the PL-located accelerators and the PS-located OS, such as `io_uring` and virtIO. In these cases, the previously reported measurements only represent a performance upper limit as the protocols' control logic must also be considered. The exact final timings are difficult to predict as they are influenced by orthogonal factors such as the communication protocol's control logic, the PL frequency, bus concurrency, etc. Implementing and evaluating such OS-to-accelerator communication is part of the authors' future research.

VI. Conclusion and Extensions

This article presents USW; an open-source SpinalHDL library to simplify HW/SW co-design on AMD-Xilinx UltraScale+ platforms. It does so by conveniently generating ready-to-use software infrastructure, effectively supporting collaborative development on the UltraScale+ platforms.

At the time of writing, the library is in its inception, and further development and refinement will take place. The authors foresee three directions. (1) Expanding the supported interfaces and I/O types (*e.g.*, CoreSight trace port interface, Raspberry Pi camera), boards, and Vivado versions. An important milestone is to provide hardware and software support for designing cache-coherent PL accelerator in the form of (a) re-usable hardware templates and (b) generating ready-to-use Arm Trusted Firmware patches. (2) Addition of support for established AMD-Xilinx technologies such as partitioning of the PL and partial reprogramming. (3) Simplification of PS-PL communication via the generation of software support and hardware modules following established protocols such as `io_uring`, `remote_proc`, and ROS.

VII. Acknowledgments

Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research. This research was supported by the National Science Foundation (NSF) under grant number CSR-2238476, and by Red Hat Research (#2025-01-RH01). The presented demonstrators are the results of collaborations with many colleagues and students. Heartfelt thanks to Bassel El Mabsout and Patrick Carpanedo for making the *"On-the-fly AES encryption and Decryption"* a reality and to Andres Rodrigo Zapata Rodriguez, Lukas Neef, and Joey Dihardjo for the realization of the *"Inverted pendulum"*. Finally, many thanks to Francesco Ciarolo for the support, brainstorming sessions, and early technical contributions.

References

[1] C. Intel, "Intel's stratix 10 fpga: Supporting the smart and connected revolution," https://newsroom.intel.com/editorials/intels-stratix-10-fpga-supporting-smart-connected-revolution/, October 2016, accessed on 19.01.2022.

[2] M. M. T. Inc., "Polarfire soc - lowest power, multi-core risc-v soc fpga," https://www.microsemi.com/product-directory/soc-fpgas/5498-polarfire-soc-fpga, July 2020, accessed on 09.01.2020.

[3] G. Alonso, T. Roscoe, D. Cock, M. Ewaida, K. Kara, D. Korolija, D. Sidler, and Z. ke Wang, "Tackling hardware/software co-design from a database perspective," in *Conference on Innovative Data Systems Research (CIDR)*, Amsterdam, Netherlands, Jan. 2020.

[4] T.-J. Chang, A. Li, F. Gao, T. Ta, G. Tziantzioulis, Y. Ou, M. Wang, J. Tu, K. Xu, P. J. Jackson, A. Ning, G. Chirkov, M. Orenes-Vera, S. Agwa, X. Yan, E. Tang, J. Balkind, C. Batten, and D. Wentzlaff, "Cifer: A 12nm, 16mm2, 22-core soc with a 1541 lut6/mm2 1.92 mops/lut, fully synthesizable, cachecoherent, embedded fpga," in *2023 IEEE Custom Integrated Circuits Conference (CICC)*, 2023, pp. 1–2.

[5] Xilinx, "Zynq ultrascale device technical reference manual," https://docs.xilinx.com/r/en-US/ug1085-zynq-ultrascale-trm/Zynq-UltraScale-Device-Technical-Reference-Manual, September 2022, accessed: 08.11.2022.

[6] Y. Tan, Y. Zhu, Z. Huang, H. Tan, and W. Chang, "Brief industry paper: Real-time image dehazing for automated vehicles," in *2023 IEEE Real-Time Systems Symposium (RTSS)*, 2023, pp. 478–483.

[7] F. Restuccia and A. Biondi, "Time-predictable acceleration of deep neural networks on fpga soc platforms," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 441–454.

[8] F. Restuccia, M. Pagani, A. Biondi, M. Marinoni, and G. Buttazzo, "Modeling and Analysis of Bus Contention for Hardware Accelerators in FPGA SoCs," in *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, vol. 165, Dagstuhl, Germany, 2020, pp. 12:1–12:23.

[9] M. Mattheeuws, B. Forsberg, A. Kurth, and L. Benini, "Analyzing memory interference of fpga accelerators on multicore hosts in heterogeneous reconfigurable socs," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 1152–1155.

[10] F. Restuccia, A. Biondi, M. Marinoni, G. Cicero, and G. Buttazzo, "Axi hyperconnect: a predictable, hypervisor-level interconnect for hardware accelerators in fpga soc," in *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, ser. DAC '20. IEEE Press, 2020.

[11] M. Pagani, E. Rossi, A. Biondi, M. Marinoni, G. Lipari, and G. Buttazzo, "A Bandwidth Reservation Mechanism for AXI-Based Hardware Accelerators on FPGAs," in *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, vol. 133, Dagstuhl, Germany, 2019, pp. 24:1–24:24.

[12] A. Biondi, A. Balsini, M. Pagani, E. Rossi, M. Marinoni, and G. Buttazzo, "A framework for supporting real-time applications on dynamic reconfigurable fpgas," in *2016 IEEE Real-Time Systems Symposium (RTSS)*, 2016, pp. 1–12.

[13] J. Goossens, X. Poczekajlo, A. Paolillo, and P. Rodriguez, "Acceptor: a model and a protocol for real-time multi-mode applications on reconfigurable heterogeneous platforms," in *Proceedings of the 27th International Conference on Real-Time Networks and Systems*, ser. RTNS '19. Association for Computing Machinery, 2019, p. 209–219.

[14] S. Roozkhosh, D. Hoornaert, and R. Mancuso, "Caesar: Coherence-aided elective and seamless alternative routing via on-chip fpga," in *2022 IEEE Real-Time Systems Symposium (RTSS)*, 2022, pp. 356–369.

[15] S. Roozkhosh and R. Mancuso, "The potential of programmable logic in the middle: Cache bleaching," in *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2020, pp. 296–309.

[16] D. Hoornaert, S. Roozkhosh, and R. Mancuso, "A Memory Scheduling Infrastructure for Multi-Core Systems with Re-Programmable Logic," in *33rd Euromicro Conference on Real-Time Systems (ECRTS 2021)*, vol. 196, Dagstuhl, Germany, 2021, pp. 2:1–2:22.

[17] I. Izhbirdeev, D. Hoornaert, W. Chen, A. Zuepke, Y. Hammad, M. Caccamo, and R. Mancuso, "Coherence-aided memory bandwidth regulation," in *IEEE Real-Time Systems Symposium (RTSS)*, 2024.

[18] C. Papon and Y. Xiao, "Spinalhdl," https://github.com/SpinalHDL/SpinalHDL, accessed: 16.04.2025.

[19] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. Association for Computing Machinery, 2012, p. 1216–1225.

[20] F. Kermarrec, S. Bourdeauducq, J.-C. L. Lann, and H. Badier, "Litex: an open-source soc builder and library based on migen python dsl," 2020.

[21] D. Hoornaert. Ultrascale spinalhdl wrapper. [Online]. Available: https://github.com/denishoornaert/ultrascale-spinal-wrapper

[22] AMD, "Ug1399: Vitis high-level synthesis user guide," https://docs.amd.com/viewer/book-attachment/o4VR_92ERnsC86VNmOmjrw/dUOl6pD9nb5aE0UIQbrzNA-o4VR_92ERnsC86VNmOmjrw, July 2023, accessed: 16.04.2025.

[23] L. Josipovic, A. Guerrieri, and P. Ienne, "Synthesizing general-purpose code into dynamically scheduled circuits," *IEEE Circuits and Systems Magazine*, vol. 21, no. 2, pp. 97–118, 2021.

[24] C. Papon. Axi4slavefactory. [Online]. Available: https://github.com/SpinalHDL/SpinalHDL/blob/dev/lib/src/main/scala/spinal/lib/bus/amba4/axi/Axi4SlaveFactory.scala

[25] S. Roozkhosh, B. El Mabsout, C. Rodrigues, P. Carpanedo, D. Hoornaert, S. Min Tan, B. Lubin, M. Caccamo, S. Pinto, and R. Mancuso, "Burning fetch execution: A framework for zero-trust multi-party confidential computing," in *GenZero 2024*, to appear.

[26] D. Hoornaert. Aes in spinalhdl. [Online]. Available: https://github.com/denishoornaert/AES

[27] L. Neef and D. Hoornaert. Inverted pendulum. [Online]. Available: https://github.com/denishoornaert/InvertedPendulum

[28] S. W. Min, S. Huang, M. El-Hadedy, J. Xiong, D. Chen, and W.-m. Hwu, "Analysis and optimization of i/o cache coherency strategies for soc-fpga device," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 301–306.