Hardware Isolation Support for Low-Cost SoC-FPGAs

Daniele Passaretti, Felix Böhm, Martin Wilhelm and Thilo Pionteck

September 13, 2022

Hardware Isolation Support for Low-Cost SoC-FPGAs

Daniele Passaretti, Felix Boehm, Martin Wilhelm, and Thilo Pionteck

Institut für Informations- und Kommunikationstechnik,
Fakultät für Elektrotechnik und Informationstechnik,
Otto-von-Guericke-Universität Magdeburg, Germany
`{name.surname}@ovgu.de`

Abstract. In the last years, System-on-Chip (SoC)-FPGAs have been widely used in Mixed-Criticality Systems, where multiple applications with different criticality domains are executed. In these systems, it is essential to guarantee isolation between the associated memory regions and peripherals of different application domains. Most high-performance SoC-FPGAs already provide hardware components for supporting isolation. By contrast, low-cost SoC-FPGAs usually don't have any mechanism for guaranteeing isolation. In this paper, we investigate the problem of hardware spatial isolation in low-cost SoC-FPGAs. First, we point out the issues and the limitations given by the fixed components in the Processing System and show how to address them. Second, we propose a Protection Unit, which is a lightweight hardware architecture for AXI communication that ensures memory and peripheral isolation between masters of different protection domains. The proposed architecture can be instantiated either on the master or on the slave side of an AXI interconnection. In addition, it is scalable from 1 to 16 memory regions, and application domains and policies are set up at run-time. We implement our architecture on the SoC-FPGA XC7Z020, where a Microblaze soft-core and the Arm Cortex-A9 are used simultaneously for different application domains. In the proposed implementation, the Protection Unit is implemented in combinatorial logic, and its execution does not contribute to the critical path. Therefore, it adds zero latency for the single communication transaction and uses only 0,5 % lookup tables and 0,1 % flip-flops of the target SoC-FPGA.

Keywords: Hypervisor · Mixed-Criticality Systems · Hardware/Software Co-Design · Edge Computing · Confidential Computing

1 Introduction

With the advent of Industry 4.0 and new computational paradigms, such as Internet-of-Things and Edge Computing, the focus of real-time processing is shifting from the cloud to sensors, creating the demand for small, scalable and energy-efficient processing devices [5,17]. These challenges are frequently tackled through the utilization of System-on-Chips (SoCs) that integrate a Field Programmable Gate Array (FPGA). In many fields, e.g. driver assistance [7], medical [19,18], railway, or avionic systems, these must adhere to various security and dependability requirements. At the same time, the same component should often be used for the execution of different applications. When multiple tasks of different application domains must coexist in a system, it is essential to guarantee spatial and temporal isolation between them, such that they cannot interfere with each other. This is especially important in Mixed-Criticality Systems (MCS), where tasks with different criticalities are run together in one system [6].

Spatial isolation protects shared peripherals or memory regions so that tasks can access only a part of them freely, or only a subset of existing tasks can access them [22]. Temporal isolation protects shared peripherals or memory regions so that only a subset of tasks can access them in a given time [22].

In a SoC-FPGA, the FPGA is also referred to as the Programmable Logic (PL), whereas the hardwired part, which contains the Application Processing Unit (APU), the fixed communication infrastructure, hardwired peripherals, and all other fixed components, is called the Processing System (PS). For high-performance SoC-FPGAs, vendors usually provide proprietary solutions for the isolation problem as part of the PS [8,15,23] through protection units. In contrast, low-cost SoC-FPGAs, which are mostly used close to sensors for Internet-of-Things applications, do not have such solutions provided for isolation. In addition, they have strict limitations in terms of available resources. Hence, the realization of a robust and flexible isolation mechanism on low-cost SoC-FPGAs is still an open challenge.

Various solutions have been proposed in literature for generic SoC-FPGAs [10,13,14]. While these solutions can be adapted for low-cost systems, most of them use protection units that rely on external memory to implement the mechanisms to check access policies. Due to the resource limitation of low-cost SoC-FPGAs and the usage of external memory, these solutions can lead to resource problems and severe timing issues. In our work, we point out the isolation challenges caused by the PS in modern low-cost FPGAs and propose a new isolation method to resolve these issues.

In our method, we statically associate the masters of the different application domains with one or more protection domains (PDs) and the available memory/peripheral space addresses with memory regions (MRs). Then, at run-time, dynamic access policies (APs) are set in order to spatially and temporally isolate masters associated with different PDs and/or MRs, similar to a white list. With this separation, it is possible to implement the whole decision path for granting AXI transactions completely with combinatorial logic and to guarantee isolation through a lightweight Protection Unit (PU). Using this approach, the proposed PU has an execution time that does not contribute to the critical path and can be added to the system between two AXI-Interfaces without additional latency.

The PU is described in SystemVerilog and is implemented to support up to 16 PDs and 16 MRs per instance. In addition, it is highly flexible in the sense that it can be instantiated in various parts of the communication infrastructure and deployed on any SoC-FPGA where the communication infrastructure uses the AXI-Interface. This flexibility enables a degree of optimization that is essential for low-cost SoC-FPGAs. We validate the design using a simulation environment together with a test design running on the SoC-FPGA XC7Z020 at an AXI clock speed of 100 MHz.

The rest of the paper is structured as follows: Sec. 2 describes related works from Industry and Academia, and Sec. 3 discusses the isolation limitations in low-cost SoC-FPGAs. Sec. 4 describes the proposed isolation method and the lightweight PU with its micro-architecture, a functional example, and validation. Sec. 5 presents the implementation and integration of the PU in the target low-cost SoC-FPGA. Sec. 6 discusses the advantages of the proposed method and compares it to related works.

2 Related Work

Supporting isolation is a well know problem in both academia [10,13,14,21] and industry [8,15,20,23]. As mentioned above, most of the proposed methods are designed for high-performance SoC-FPGAs. In this paper, we mainly consider related works, where enough information is given to compare them with our work in terms of functionality and performance.

In industry, one of the most used mechanisms to guarantee isolation in SoC-FPGAs is the Arm TrustZone [20]. It's a system-wide security extension, which provides two execution contexts, *secure* and *non-secure*. Memory regions and peripherals can be configured to allow access to only the secure context. On the downside, the Arm TrustZone considers only two domains and cannot be extended. Furthermore, several vulnerabilities and weaknesses have been investigated in [20], which are the consequence of the lack of robust Trusted Execution Environments (TEE) runtime implementations, and microarchitectural defects.

Due to these limitations, Xilinx proposed two types of protection units for high-performance SoC-FPGAs: the Xilinx Peripheral Protection Unit (XPPU), and the Xilinx Memory Protection Unit (XMPU) [23]. Both protection units are hardwired and implemented in the Processing System (PS) of Ultrascale+ MPSoC chips. They check the master ID and the accessing address for each AXI transaction inside the PS and between PS and PL to guarantee spatial isolation. If the transaction is allowed, then it proceeds normally; otherwise, it's invalidated and won't reach its intended destination. XMPU and XPPU have two different implementations because they are optimized for memory read/write and for device control, respectively. These protection units are not implemented in low-cost SoC-FPGAs.

Sensaoui et al. [21] propose a hardware architecture for isolation in low-cost SoCs, which is for ASIC. It is compatible with RISC-V and Arm CPUs and considers the bus communication between the different masters and memories, for which different policies are defined. The hardware architecture is mainly composed of a master "look-side buffer" and their "uCode" block that implements the logic responsible for checking the communication transactions.

Kumar Saha and Bobda [11] propose a security framework for isolation, which uses Mandatory Access Control (MAC) based authentication policies. It considers a fixed subpart of the device ID and the memory address for defining different application domains and memory regions. This solution can be used to protect the different peripherals in the PL. It consists of software and hardware management modules; therefore, it is suitable only for SoC-FPGAs. The software manages the policy server that is implemented on the PS, and sets up the Hardware IP management module (HIMM). These are implemented on the PL in front of each peripheral (slaves in the communication protocol). This method requires a Linux OS to run the software module, and the policies are read at run-time from the PS. Due to the delay in reading the policies, this solution adds to the transaction a delay of 35 microseconds if a new entry has to be fetched.

LeMay et al. [14] propose a hardware-based Network-on-Chip Firewall (NoCF) that also uses a software and a hardware module. The software configures NoCF and specifies the policies to enforce isolation between the cores. The policies are maintained by an integrity kernel that runs on a dedicated integrity core which is implemented in hardware. The software module can be controlled by a hypervisor, but the overall NoCF is mainly optimized for NoC communication infrastructure and uses a big amount of FPGA resources, and it is not suitable for Low-Cost SoC-FPGAs in most of the cases.

Kornaros et al. [10] propose a memory partition protection unit that isolates physical memory regions by applying access rules. This unit is implemented in the PL and is set up by a software driver running on the PS. The driver runs on Linux. The Memory Partition Protection Unit (MPPU) supports a maximum of 16 application domains. This solution can also be used for Low-Cost SoC-FPGAs but with a lack of performance, as we will discuss in Sec. 6.

All analyzed works consider different methods for implementing the policy rules and policy checking. Yet, they are not flexible for deploying them in different parts of the system design, e.g., close to the master side or to the slave side. We will compare them with the proposed isolation method and the proposed PU, in Sec. 6.

3 Isolation limitations in low-cost SoC-FPGAs

In this section, we describe the micro-architecture of low-cost SoC-FPGAs, analyze their limitations in terms of spatial isolation, and propose our solution for isolating the peripherals on the PS.

As mentioned in Sec. 1, SoC-FPGAs generally consist of the PS, containing hardwired processing units, input / output peripherals and communication infrastructure, and the PL, comprising programmable look-up tables, memory, and digital signal processing components. To describe the micro-architecture of low-cost SoC-FPGAs, we analyzed the Xilinx Zynq-7000 [3] and Intel Cyclone V SoC [9], which are the low-cost SoC-FPGAs of the two biggest FPGA vendors in the world. As shown in Fig. 1, these two FPGAs have a similar micro-architecture at the system level and use the same PS interconnection core link, which is the Arm NIC-301 [4]. To highlight the similarity between the two chips, we use the AMD-Xilinx nomenclature in Fig. 1, even though AMD-Xilinx and Intel-Altera use different names in their reference manuals.

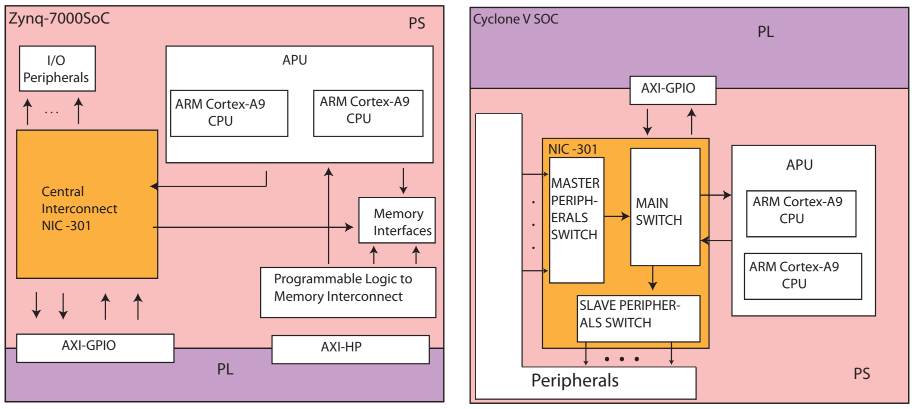
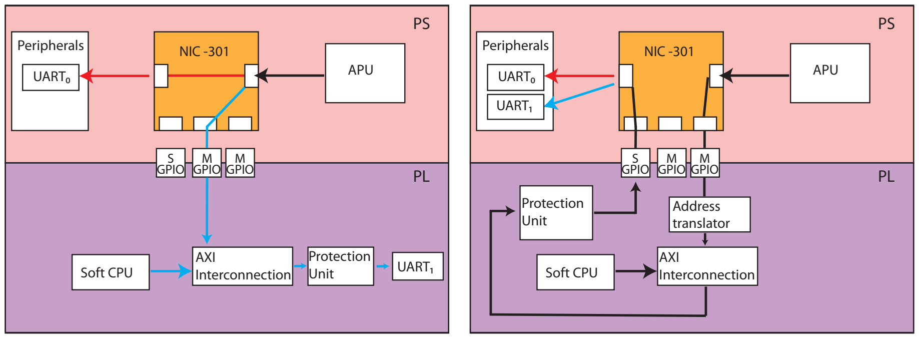


Fig. 1: Micro-architecture of Zynq-7000 SoC and Cyclone V SoC

As shown in Fig. 1, the Application Processing Unit (APU) is connected to the PS peripherals through the NIC-301, and there are no hardware protection mechanisms between them. Hence, malicious applications running on the APU that try to access PS peripherals can not be blocked, and spatial isolation can not be guaranteed. Due to the fact that PS components are dedicated physical components with a fixed implementation, it is not possible to add any protection units between the APU and PS peripherals. As shown by the red arrow in Fig. 2a the APU has direct access to PS peripherals without any checks. For this reason, the PS peripherals usually are exclusively used by the APU, and the connection between PS peripherals and the PL is not initialized. In addition, if a peripheral should be shared between the APU and a PL master (e.g. a Microblaze), it has to be instantiated on the PL where can be protected by a protection unit. This design is shown in Fig. 2a, where UART_0 has exclusive access to the APU and UART_1 is shared between the APU and the Soft CPU.



(a) Default communication pattern (b) Proposed communication pattern

Fig. 2: State-of-the-art and proposed communication pattern between APU, PL and PS peripherals in low-cost SoC-FPGAs

The design solution in Fig. 2a guarantees the isolation of UART_1, with the additional cost of implementing the UART peripheral on the PL.

In this work, we propose a communication pattern that allows sharing of PS peripherals between PS and PL masters while guaranteeing their isolation through a protection unit. In the proposed communication pattern, the APU's direct access to PS peripherals over the NIC-301 is disabled. Instead, PS peripherals are mapped into the APU address space in the same way that GPIO peripherals are mapped into the PL address space, as shown in Fig. 2b. By this, the NIC-301 forwards the AXI transactions from the APU to the PL through the PS-to-PL Master interface. The transactions cross the AXI interconnect bus which forwards them to the PS peripheral through the protection unit and the PL-to-PS interface, which is connected to the NiC-301. By sending all transactions between APU and PS peripheral over the PL, we can instantiate a protection unit that guarantees spatial isolation of the PS peripherals and can also be accessed from multiple masters, as shown in Fig. 2b. In this way, we have guaranteed isolation and also reduced the PL resources' utilization.

To correctly forward AXI transactions from APU to PL and back from PL to PS peripherals, the following assumptions must be fulfilled:

– PS peripherals can be accessed with their physical addresses only from the PL. To guarantee that the APU can not access to PS peripherals, they are not enabled in the PS. They are mapped only to the PL GPIO master interface instead.
– The PS peripherals are mapped to the APU as GPIO peripherals. In this way, the GPIO space address represents the virtual address of the PS peripherals, and the NIC-301 will forward all AXI transactions from APU to PL over the PS-to-PL interface.
– The AXI transaction coming from the PS that has the GPIO base address must be translated to the physical address of the PS peripherals and forwarded to the AXI interconnect component. For that, an AXI address translator is required.
– The AXI interconnect, which is connected to the AXI address translator, has to be connected to the PS peripherals through the PS-to-PL GPIO interface. In this way, the transactions can be forwarded to the PS peripherals
– A protection unit has to be instantiated between the AXI Interconnect and the PS-to-PL GPIO interface. In this way, the instantiated protection unit checks all transactions and guarantees isolation of the PS peripherals.

For the best of our knowledge, the proposed communication pattern has not been discussed in other related works that simply assume that PS peripherals are accessible only by the APU. But reserving PS peripherals only for APU accesses can lead to an under-utilization of the PS peripherals and an over-utilization of the PL, where additional peripherals have to be instantiated, instead. In low-cost FPGAs the distribution of resources is essential, therefore this peripheral distribution can limit the overall system design. We tested the utilization of PS peripherals from the PL side with a soft-processor as master and the protection unit without sharing the peripherals with the APU. The details on the proposed isolation method, the protection unit, and its features and configuration will be presented in detail in the following section.

4 Proposed method

This section clarifies our terminology, presents our proposed method for supporting isolation based on Protection Domains, Memory Regions, and Access Policies, and describes the architecture and validation of the introduced Protection Unit.

In our work, a **Protection Domain** (PD) consists of a set of master components of the same application domain. A master component can be part of multiple PDs at the same time. A PD is defined by a *domain ID* and a *domain mask*. These two parameters are used to derive the association of its master with one or more PDs in the AXI transaction. This is possible because each AXI transaction contains an AXI ID, which is related to the master that generates the transaction.

Table 1: Example of Domains and AXI ID relations

	Domain Mask	Domain ID	exemplary IDs
Domain 0	1100	$\underline{10}00$	$\underline{10}11$, $\underline{10}00$
Domain 1	1110	$\underline{100}0$	$\underline{100}0$
Domain 2	1110	$\underline{101}0$	$\underline{101}1$

The matching between an AXI ID and its PDs is done by comparing the bits of the available domain IDs, which are selected by the mask, to the corresponding bits of the transaction's AXI ID. If all of the bits are equal, the transaction belongs to the current PD. Table 1 shows a simple example, where the master ID *1011* is part of the domains 0 and 2. To associate multiple masters with one of the PDs, the designer has to extract the common subpart of the master IDs to determine the domain ID. To make this step easy and to determine a domain ID, when there are no common subparts between different master IDs, we propose an ID-manipulating component, which adds or modifies the AXI-ID. It will be described in Sec. 5. The main advantage of this solution is that its implementation requires only few resources to match the master component ID with the domain IDs at run-time. In particular, it is possible to determine the domain associations of each AXI transaction with simple combinatorial logic.

A **Memory Region** (MR) is an aligned address space. A peripheral and a memory address can be part of multiple MRs, in the same way that masters can be part of multiple PDs. We define a MR by using a starting address and its most significant bits, which contain the addresses of the whole MR. Consequently, a MR always consists of continuous addresses. To optimize the execution time and resource utilization of the MR matching component, we use the position of the Least Significant Bit (LSB) of the MR as a parameter. The LSB position determines the size of the memory region that is equal to $2^{LSB\ pos.}$. By using it as a parameter, the PU has to check only the range of bits between the MSB and the LSB per input address. An example of MR matching is given in Sec. 4.1.

An **Access Policy** (AP) describes which protection domain may access which memory region, i.e., it consists of a set of "PD_X may read/write MR_Y" rules. A transaction may pass if any of these rules gives it permission to do so. If there is no rule for a master because it is not associated with any PD or MR, all its requests will be denied. In this work, each protection unit has their own dedicated APs for read and write access.

PDs, MRs and their parameters are set at design time, while APs are defined at run-time. In this way, it is possible to optimize each PU instance at synthesis time and implement the whole decision path as a fixed combinatorial logic component, where the input is the AXI-transaction and the rules of the AP. Only the APs are set at run-time; otherwise, temporal isolation can not be guaranteed.

4.1 Protection Unit Architecture

The Protection Unit (PU) restricts traffic through an AXI connection based on the given policy. An overview of the architecture is shown in Fig. 3. A PU has three AXI instances. The two red interfaces are complementary, pass-through, full AXI interfaces and identically parameterized. These are transparent for all allowed transactions. The orange AXI-Lite interface provides access to status, control, and policy configuration registers.

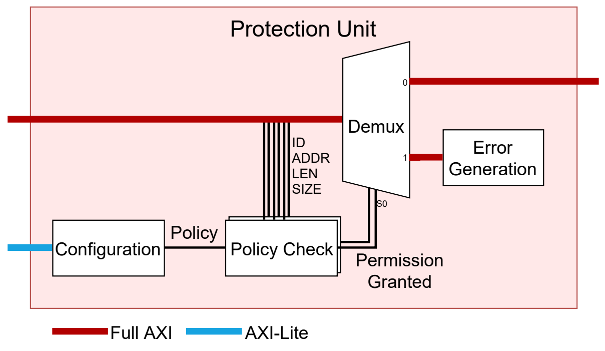


Fig. 3: Architecture of the Protection Unit

The APs are stored in a configuration module in the PU. Each PU contains its own APs, one for reading and one for writing transactions. This module forwards the policies as signals to the Policy Check module that is the core of the PU. In this way, every clock cycle, all current APs are read from the Policy Check. This last component is responsible for matching the master ID (AXI-ID) and the incoming memory address with the PDs and the MRs available. Then it checks the defined policy and the signal's permission to read/write. In each PU, there are two instances of the Policy Check module, one for the read channel and the other for the write channel.

As shown in Fig. 4, the Policy Check contains PD matching and MR matching components. These two components are instantiated once for each PD and

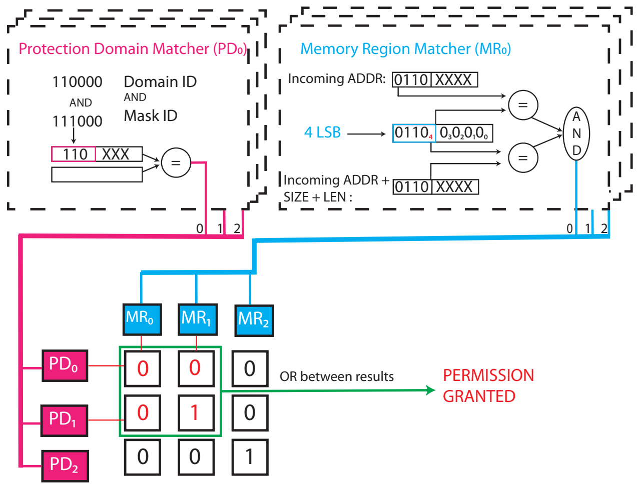


Fig. 4: Policy check functionality

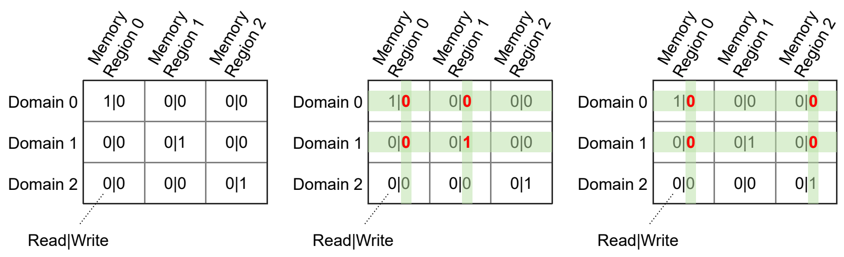
MR, respectively. They check the AXI ID of the transaction for corresponding domains and the incoming address for corresponding memory regions. Matched domains and memory regions are then used to determine which entries in the policy to check. In the example, shown in Fig. 4, the input address is associated with MR_0 and MR_1, and the AXI-ID with PD_0 and PD_1. Because the AP for PD_1 and MR_1 is equal to 1, the transaction is granted. To sum it up, the Policy Check grants or denies permission based on the PDs and address ranges, current APs and the ID, ADDR, LEN and SIZE signals of the transactions.

The decision of the Policy Check is sent to an AXI demultiplexer which forwards the transaction. If the transaction is granted, it is forwarded downstream to its intended destination. If the transaction is denied, it is forwarded to an internal error-generating slave, which asserts errors as defined by the AXI standard without having a stall in the communication. For the implementation of the prototype, both the AXI demultiplexer and the error generator are taken from the PULP-platform AXI library [12].

4.2 Example

Here, we describe a concrete example of an Access Policy to give a better idea of what they look like and how a decision is inferred. For simplicity, a system with two masters, two slaves, and a single PU is considered.

The first master is included in the PD_1 and the second one in the PD_2. PD_0 includes both. Analogously, the first slave is associated with MR_1, the second one with MR_2, and both are part of MR_0. For access to slaves, masters must be included in PDs, and slaves must associate their memory space address to MRs. These parameters have to be set before synthesis. During run-time, read access is given to PD_0 for the MR_0, in this way both masters can read from to both slaves. Instead, write access is given to PD_1 for the MR_1 and PD_2 for the MR_2. This results in the APs shown in Fig. 5a



(a) Example Policy (b) Access given (c) Access denied

Fig. 5: Exemplary Policy Configuration and Decision

If the first master tries to write on the first slave, PD_0 and PD_1 are matched as well as MR_0 and MR_1 are matched as shown by green boxes in Fig. 5b. The matched PDs and MRs represent the eligible PDs and MRs that are considered for making the decision. These entries are marked red in Fig. 5b. In this specific case, the access is given because the policy states that PD_1 has write access to MR_1, indicated by the red 1. If the first master tries to obtain write access on the second slave, MR_0 and MR_2 are matched instead of MR_0 and MR_1, as shown in Fig. 5c, but in the AP there is no value equal to 1, so the access is denied. No access is given if none of the rules in the policy allows it.

4.3 Validation

Before integrating the PU with a running design, it has been tested in simulation, using Xilinx Vivado 2020.2 and the AXI Verification IP (VIP) [2]. The VIP is used for generating traffic on master and slave sides as well as for checking the AXI protocol compliance.

Three scenarios have been considered. In the first one, two masters, one PU, and one slave have been instantiated. The first master has been used for the run-time configuration of the PU, and the other master communicates to the slave through the PU itself. In this way, we have tested the correct functionality of the PU, when correct and illegal transactions are generated from the master to the slave. In the other two cases, we increase the masters, the PUs, and the slave to validate the behavior of the system when multiple requests arrive from different masters. In the second configuration, we have instantiated the PU on the master side (as shown in Fig. 7a), while in the third, we have instantiated them on the slave side (as shown in Fig. 7b). In all these scenarios, the generated traffic consists of typical transactions, edge cases, and random traffic. The traffic has been checked for being compliant with AXI and the configured policy.

5 SoC-FPGA Implementation

In this section, we describe the implementation of the hardware and software modules for integrating the PU in the target SoC-FPGA C7Z020 and for setting the desired APs. In addition, we introduce the AXI-ID Manipulator, which is an additional hardware component required for supporting masters/slaves that do not have defined AXI-IDs.

The PU and the AXI-ID Manipulator, as well as the internal modules, which are taken from the PULP-platform AXI library [12] are implemented in SystemVerilog. To simplify the integration of these components in a system design, we pack them as IP Core modules compatible with any Xilinx FPGAs of the 7 series family [1]. The designer can add them to the IP repository and instantiate them either in HDL entities or into a block design. As explained in Sec. 4, each PU has an AXI-Lite interface used for setting up the policies. In the system design, the AXI-Lite interface of all PUs is connected to a dedicated control bus as a slave peripheral. The master that starts up the whole system (setting up the PS and programming the PL) and manages the temporal domains at run-time is the only master which is physically connected to this control bus.

For evaluating the PU, we implemented the use case shown in Fig. 7a, which resulted in the block design shown in Fig. 6. This block design has two masters: the Cortex-A9 and a Microblaze. The Cortex-A9 is the APU of the target SoC-FPGA, and the Microblaze is a soft processor, which is instantiated in the PL. In addition, two slaves in the PL are instantiated, and the Microblaze can also access the slaves that are on the PS part. For this design, where there are two masters and multiple slaves, we instantiate the PUs on the master side. By this, we only require two PUs, as shown in Fig. 6; other possible configurations are explained in Sec. 6.

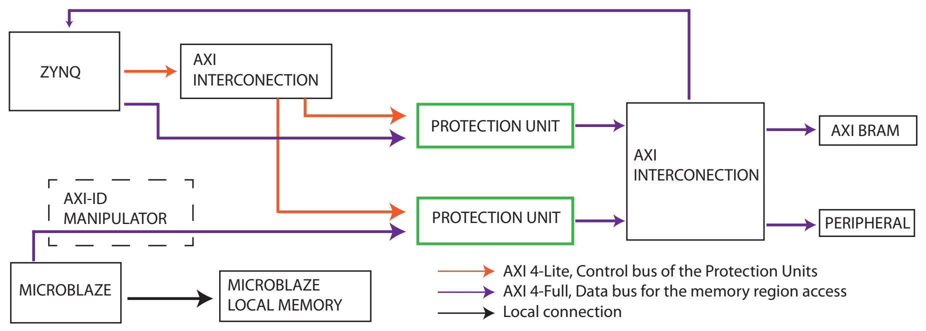


Fig. 6: Implemented system design. Two Protection Units on the master side. The AXI-ID Manipulator is not required with Microblaze, but is inserted for demonstration purpose.

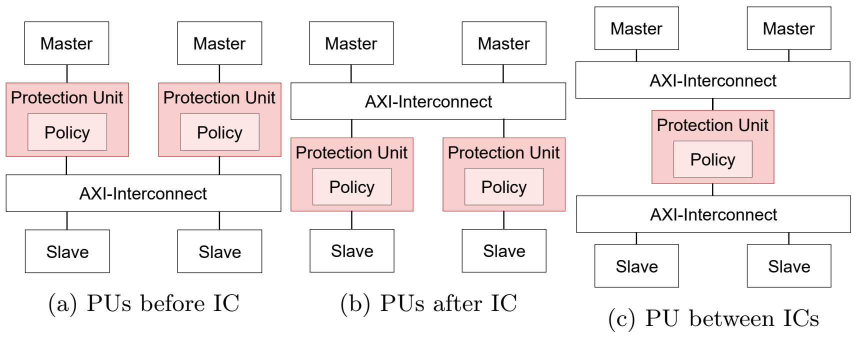
In our design, the APU is responsible for starting up the system. When the FPGA is turned on, the Cortex-A9 runs the First Stage Boot Loader (FSBL), a bare-metal program responsible for loading the bitstream from memory and programming the PL. When the PL is running, the FSBL program sets up the PUs and runs the APU and Microblaze applications. It is essential that the PUs are set up before any other applications. By this, the applications can run either as bare-metal source code or through an operating system, and the PU is set up in a secure way. In order to provide temporal isolation, the APs have to be properly updated at run-time with the support of a hypervisor. In addition, to guarantee the security of the PUs and of the whole system, the FSBL should be run with a secure boot, or the PUs should be set with the support of a hypervisor. In this paper, we focus on the proposed isolation methodology and the PUs, therefore we do not explain all security aspects that involve the whole system.

During the implementation, the designer should know all AXI-IDs of masters and slaves. If an AXI-ID is not provided, or a master/slave has no AXI-ID, the designer can use the AXI-Manipulator between the master/slave module and the AXI interconnection bus as shown in Fig. 6. The AXI-ID Manipulator adds or overwrites specific bits of the AXI-ID for the AXI requests and restores the old AXI-ID for the related responses. This component is essential for defining the various AXI-IDs used for recognizing the different PD.

6 Results and Discussion

In this section, we report and discuss the results of our work. We consider various implications related to the deployment step. In addition, we analyze the results of the testing design and different PU configurations used for comparing our work with the related works. In the end, we point out possible improvements and potential future works.

The proposed isolation method aims for low-cost SoC-FPGAs, so it has been conceived to be flexible and lightweight. The presented PU can be deployed on every SoC-FPGA that uses AXI as communicating interface protocol. As shown in Fig. 7, it can be instantiated on the master side, slave side, and between two AXI interconnects. This allows the PUs to be deployed at the points in the system where they are most effective. These points may differ between different systems.



(a) PUs before IC (b) PUs after IC (c) PU between ICs

Fig. 7: Exemplary Deployment of Protection Units

A system design, where masters are much more than slaves can save resources if the PUs are deployed on the slave side, or the other way around. When the PU is instantiated on the master side, it manages only the transaction coming from this master component; in this case (shown in Fig. 7a), each master component has its PU, where only the MRs have to be checked. This configuration can also have the advantage of blocking unwanted transactions before they could cause contention in the downstream interconnect.

In contrast, if the PU is instantiated on the slave side, each slave has its own PU, which is only responsible for the MRs of the related slave. The last configuration is shown in Fig. 7c, where there is only one PU. In this case, two interconnect busses are used, where the first groups all master connections and the second groups all slave connections. In this case, all PDs and MRs are managed by a single PU. This solution can be advantageous when there is a similar number of masters and slaves in the system and they are few, because it will use less resources than using multiple PUs.

To analyze the performance of our PU, we have implemented it with Vivado 2020.2 on the target SoC-FPGA XC7Z020 using different configurations. First, we run it in the test design shown in Fig. 6. In this design, we use two PUs and we set the AXI Clock to a frequency of 100 MHz. Each PU has one PD and two MRs.

Table 2: Resource utilization for the test design

	LUTs	FFs	LUTRAM	Block	DSP
PU: axi demux	53	22	0	0	0
PU: configuration block	58	109	0	0	0
PU: policy check write	35	0	0	0	0
PU: policy check read	34	0	0	0	0
PU: axi error	22	15	0	0	0
PU Microblaze	202	146	0	0	0
Test Design	4470	4449	299	3	3

As shown in Table 2, the PU in front of the Microblaze uses only 202 LUTs and 146 FFs. In fact, in relation to the resource utilization of the test design, it uses only 4.51% of the LUTs and 3.28% of the FFs. Also, in relation to the available resources of the target FPGA, it uses only 0.37% of LUTs and 0.13% of FFs. This low utilization of resources is essential for low-cost SoC-FPGAs with limited resources. Table 2 also shows the resources used by the internal components. As expected, the policy check components do not use any FFs, because it has been designed with combinatorial logic in the Register-Transfer Level (RTL). This is an important result because this component determines the decision path and the delay of the granted transaction, which is combinatorial and does not contribute to the critical clock path. Since we set the clock frequency at 100MHz, it means that the PU processes a single transaction in less than 10 ns. Most of the FFs are used in the configuration block that contains the control/status register and APs of the PU.

Table 3: Resources' utilization and execution time (c.c. means clock's cycle)

Work	LUTs	FFs	LUTRAMs	Execution Time
PU, 1PD/16MR	339	198	0	<10 ns (<1 c.c.)
PU, 16PD/1MR	191	198	0	<10 ns (<1 c.c.)
PU, 16PD/16MR	950	678	0	<10 ns (<1 c.c.)
PU, 1PD/1MR	164	168	0	<10 ns (< 1 c.c.)
MPPU [10]	655	1082	12	(4 c.c) only the decision
HIMM [11]	86	75	6	220-35000 ns

In Table 3 we compare the resource utilization of our PU to the MPPU of Kornaros et al. [10] and the Hardware IP management module (HIMM) of Kumar Saha and Bobda [11].

Our PU's resource usage is listed for different numbers of PDs and MRs. Since the resource usage is optimized at design time based on the set parameters, we can see how the PU scales by changing the number of PDs and MRs. The MPPU has similar functionality as the PU with 1PD/16MR configuration as it is also able to differentiate 16 memory regions. Because we define PDs and MRs at design time and implement the policy check only with LUTs, we reached better performance in terms of resource optimization and execution time. The HIMM has similar functionality to the 16PD/1MR configuration as they protect a single slave by 16 different application contexts (application contexts have the same functionality as our PDs). They use less resources as most of their processing is done outside the HIMM. They manage and store the policy in the PS and hold local copies in the HIMM. If a rule is not present locally, it has to be fetched first leading to delays of up to 35000 ns.

In the future, additional optimization can be done by substituting the library components with specialized components that are optimized for the PU. In addition, this PU can be used as a foundation to integrate a basic hypervisor in the SoC-FPGA that will manage the temporal isolation properly. Finally, the combinatorial path of the PU can be pipelined in the case that it can not be executed in one clock cycle on a given target FPGA.

7 Conclusion

In this work, we have analyzed the isolation limitations of low-cost SoC-FPGAs and we have proposed an isolation method that results in a lightweight Protection Unit that uses combinatorial logic for the decision path. We also implemented and integrated the PU in a test design, where we confirmed that a single transaction is processed with no additional latency. The key to this result is that the decision path is combinatorial, so no FFs are used in the policy check and it does not contribute to the critical path of the system design. Compared to the state of the art, where external memory is the bottleneck, the presented isolation method and the PU implementation result in a much faster execution. In addition, its deployment flexibility, configurability and scalability makes it especially well-suited for the integration with low-cost SoC-FPGAs. The presented Protection Unit can be found as SystemVerilog component and/or IP Core block on our website [16].

References

1. AMD-Xilinx: 7 Series FPGAs Data Sheet: Overview(DS180)
2. AMD-Xilinx: AXI Verification IP LogiCORE IP Product Guide (PG267)
3. AMD-Xilinx: Zynq-7000 SoC Technical Reference Manual(UG585)
4. ARM: CoreLink Network Interconnect NIC-301 Technical Reference Manual
5. De Donno, M., Tange, K., Dragoni, N.: Foundations and evolution of modern computing paradigms: Cloud, iot, edge, and fog. IEEE Access (2019)
6. Gracioli, e.a.: Designing mixed criticality applications on modern heterogeneous mpsoc platforms. In: ECRTS 2019 (2019)
7. Hassan, M.: Heterogeneous mpsocs for mixed-criticality systems: Challenges and opportunities. IEEE Design & Test **35**(4), 47–55 (2018). https://doi.org/10.1109/MDAT.2017.2771447
8. Intel: External Memory Interface Handbook Volume 3: Reference Material . https://www.intel.com/content/www/us/en/docs/programmable/683841/17-0/memory-protection.html, [Online; accessed 13-July-2022]
9. Intel-Altera: Cyclone V Hard Processor System Technical Reference Manual
10. Kornaros, G., et al.: Hardware support for cost-effective system-level protection in multi-core socs. In: 2015 Euromicro Conference on Digital System Design (2015)
11. Kumar Saha, S., Bobda, C.: Fpga accelerated embedded system security through hardware isolation. In: 2020 Asian Hardware Oriented Security and Trust Symposium (AsianHOST). pp. 1–6 (2020)
12. Kurth, A., Cavalcante, M., Zaruba, F.: PULP Platform. https://github.com/pulp-platform/axi (2022)
13. Kurth, A., Rönninger, W., Benz, T., Cavalcante, M., Schuiki, F., Zaruba, F., Benini, L.: An open-source platform for high-performance non-coherent on-chip communication. IEEE Transactions on Computers (2022)
14. LeMay, M., Gunter, C.A.: Network-on-chip firewall: Countering defective and malicious system-on-chip hardware. In: Logic, Rewriting, and Concurrency (2015)
15. Microchip: Polarfire® soc mss technical reference manual
16. Passaretti, D., Böhm, F., Pionteck, T.: Isolation-support for low-cost soc-fpgas. https://github.com/pasdani/Isolation-Support-for-Low-Cost-SoC-FPGAs
17. Passaretti, D., Ghosh, M., Abdurahman, S., Egito, M.L., Pionteck, T.: Hardware optimizations of the x-ray pre-processing for interventional computed tomography using the fpga. Applied Sciences **12**(11), 5659 (2022)
18. Passaretti, D., Pionteck, T.: Configurable pipelined datapath for data acquisition in interventional computed tomography. In: 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)
19. Passaretti, D., Pionteck, T.: Hardware/software co-design of a control and data acquisition system for computed tomography. In: 2020 9th International Conference on Modern Circuits and Systems Technologies (MOCAST). pp. 1–4 (2020). https://doi.org/10.1109/MOCAST49295.2020.9200273
20. Pinto, S., Santos, N.: Demystifying arm trustzone: A comprehensive survey. ACM Comput. Surv. (jan 2019)
21. Sensaoui, A., Hely, D., et al.: Toubkal: A flexible and efficient hardware isolation module for secure lightweight devices. In: 2019 15th European Dependable Computing Conference (EDCC). pp. 31–38. IEEE (2019)
22. Valente, G., Giammatteo, P., Muttillo, V., Pomante, L., Di Mascio, T.: A lightweight, hardware-based support for isolation in mixed-criticality network-on-chip architectures. ASTES (2019)
23. Xilinx, Inc.: Isolation methods in zynq ultrascale+ mpsocs application note