

Received 13 November 2025; revised 22 January 2026; accepted 23 January 2026.
Date of publication 28 January 2026; date of current version 5 February 2026.

Digital Object Identifier 10.1109/OJCOMS.2026.3658832

Dynamic Deployment of Real Time Services in Edge Computing Systems

SARA CUCCAGNA¹, MAURO FEMMINELLA^{1,2} (Member, IEEE), GIANLUCA GIUSEPPONI³,
BARBARA GUASTICCHI³, LUCA MORICONI³ (Member, IEEE),
AND GIANLUCA REALI^{1,2} (Member, IEEE)

¹Department of Engineering, University of Perugia, 06125 Perugia, Italy

²Consorzio Nazionale Interuniversitario per le Telecomunicazioni (CNIT), 43124 Parma, Italy

³ELES SpA, 06059 Todi, Italy

CORRESPONDING AUTHOR: G. REALI (gianluca.reali@unipg.it)

This work was supported in part by European Commission through European Regional Development Fund (ERDF) 2021–2027, Umbria Region, Research Program in favour of ELES Semiconductor Equipment S.p.A, and through the NextGenerationEU program, in part by the Italian Ministry of University and Research (MUR) National Innovation Ecosystem under Grant ECS00000041–VITALITY, and in part by the MUR Extended Partnerships under Grant PE00000001–RESTART.

ABSTRACT System-on-Chip (SoC) platforms combining programmable logic (PL) and general-purpose processors (PS) are increasingly used in low-latency and real-time applications running in edge computing platforms, thanks to their computing capabilities, low cost, and modest energy consumption. However, their latency performance is often not fully understood in practical deployments. To address such an issue, this paper presents an experimental study of latency and throughput performance on a Zynq-7030 SoC system, explicitly targeting communications between PS and PL implemented through AXI DMA. The analysis is based on measurements conducted on a real hardware platform. By evaluating different architectural and software design choices, this manuscript provides actionable guidance for designing low-latency PS–PS or PS–PL communications on SoC devices. The presented results highlight that latency and throughput can be protocol-limited or hardware-limited, depending on the selected communication architecture. Finally, we present and evaluate an industrial case study to demonstrate the potential of the usage of SoC devices in terms of achievable performance. Although the experimental evaluation is performed on a specific hardware platform, the methodology and conclusions apply more broadly to heterogeneous SoCs relying on shared-memory DMA and Linux-based and/or real-time execution environments.

INDEX TERMS Edge computing, system-on-chip, throughput, latency, implementation.

I. INTRODUCTION

IN RECENT years, several applications that take advantage of the possibility of being deployed in edge computing systems have been defined and developed. In fact, the strategic location of the edge allows it to achieve significant benefits compared to traditional centralized cloud-based architectures, especially in scenarios where latency, reliability, security and efficiency are critical. In fact, Edge computing allows (i) local data processing, thus reducing the time needed to make data round-trips in the cloud, (ii) sending only a part of the data considered relevant (e.g., analysis results, not raw data) to the cloud core, (iii) keeping sensitive data (e.g., biometrics, videos) on-premise, reducing the risk of breaches, (iv) reducing the energy spent for data transmission, (v) adapting devices to specific contexts (e.g., extreme temperatures, vibrations), (vi) locally managing compliance with data management

regulations, and (vii) reducing operational costs of bandwidth, storage, licenses and energy consumption.

Broadly speaking, any compute device at the edge of a networked system can be considered an “edge system,” but not all are equally effective. The choice of hardware (PCs, servers, SoCs, GPUs, etc.) depends on functional requirements, constraints, and trade-offs. In this paper we focus on optimizing the edge benefits mentioned above by making use of highly specialized, compact and low-cost platforms called System-on-Chip (SoC). In fact, they allow combining multicore software processing and FPGA-based hardware acceleration, thus leveraging the strengths of both components [8]. Such devices are designed to implement embedded systems, but they can also serve as edge nodes when they process local data before cloud transmission and enable low-latency control [25]. Examples of applications that can be

framed in this context are industrial monitoring, such as detection of defects in production, or maintenance based on the level of vibrations, intelligent surveillance systems for face and object recognition, automotive applications that implement sensor-based control systems, voice recognition, and real-time monitoring of power grids.

For such applications, the computing cores present in the platform can be used to deploy different and coexisting operating systems, such as Linux and FreeRTOS. The use of Linux is ideal for implementing front-end for managing data flows, and possibly sending the relevant results to the cloud. The use of FreeRTOS is ideal for real-time control, especially when reliability and low latency are required. In addition, since the kernel size of this operating system is a few tens of kBytes, it is ideal to be deployed in secondary cores that can be activated as needed, using an AMP configuration (Asymmetric Multi-Processing) along with Linux. The simultaneous availability of FPGAs makes it possible to accelerate operations that are known to be time-consuming. For example, applications that include machine learning components, such as convolutional neural networks [12], data analysis from thousands of sensors, data encryption using AES, SHA-x, and post-quantum algorithms.

Having all these features available is therefore strategic, also due to the low energy consumption compared to that of the computing servers typically available in the cloud core. However, implementation on these devices is far from simple in order of take advantage of all their features.

In this paper we present an orchestrated deployment approach for real-time edge computing services and show the experimental results based on a Zynq-7030 SoC. This device integrates a dual core ARM-based processing systems (PS) with a programming logic (PL) including an FPGA [8]. By the use of this device and different operating systems in it, we show different implementation strategies characterized, on the one hand, by bandwidth and latency of interactions of differentiated intra-SoC components, on the other hand by different flexibility to changing operating conditions and deployment times.

The choice of the SoC used for the experimental analysis was made on the basis of the technical features that guarantee appropriate flexibility in applying the proposed methodology, of its low cost, and relatively low energy consumption. The specific interest of the selected device derives from a series of competing implementation issues, including costs, clutter of multiple boards in confined environments, the need to dissipate heat and efficiency of resource usage avoiding overprovisioning of resources. Therefore, the focus on latency and throughput, although of great importance in industrial applications, does not completely cover the complexity of the problem.

It is also worth noting that the experimental demonstration based on this platform does not affect the generality of the methodology, that can be applied to a multiplicity of SoCs distributed by different vendors, having technical features comparable or superior to that used, albeit at a higher cost

and larger footprint. This aspect is explored in more detail in Section III.

The existing literature on this subject, presented in section III, includes some interesting analyses but has not allowed us to find the answers to all the questions relating to the opportunity to use the platform in real contexts and possibly to the best configuration among those available. Thus, we produced a comprehensive and structured benchmarking across Linux $\leftrightarrow$ FreeRTOS and PS $\leftrightarrow$ PL paths, covering GPIO, AXI4-Stream, DMA, and OpenAMP/RPMsg with actionable guidelines that map traffic characteristics (continuous vs. burst, packet size) to interface choices (GPIO vs. AXI-Stream, DMA usage), including power and real-time considerations. Finally, we implemented a concrete industrial case study demonstrating orchestration (AMP, on-demand core activation, FPGA reconfiguration) and the translation of benchmarks into system design choices.

In more detail, our specific contributions are as follows:

- 1) Detailed workflow for implementing SoC-based Edge-Computing systems for data-intensive real-time applications.
- 2) Experimental analysis of the latency and throughput performance in intra-SoC communications for different implementation solutions.
- 3) Implementation of a case study belonging to the industrial context that captures the effectiveness of the proposed solution in a challenging environment.
- 4) Identification of guidelines for selecting implementation strategies based on the experimental validation.

The paper is organized as follows. Applications for edge computing are classified in Section II, along with their typical implementation issues. In Section III, we delve into an examination of the existing literature within the domain. Moving on to Section IV, we furnish a comprehensive introduction to the foundational concepts. The implementation solutions for investigating the potential of the SoC are detailed in Section V, along with proof of concept implementations. The validation outcomes of the analyzed approaches are reported in Section VI. The key insights of our analysis are shown on Section VII, along with a sample implementation of an industrial application. Finally, Section VIII includes our closing remarks and conclusions.

II. PROBLEM DESCRIPTION

Applications for edge computing systems can be classified according to some key criteria, such as latency, limiting resources, and traffic statistics. In what follows we report a classification based on these criteria, including a specific focus on how the use of SoCs such as the Zynq-7000 family, or similar SoCs, can be justified as a suitable implementation platform for each category.

- No Real-Time: Applications with relaxed latency constraints, for which response delay in the order of seconds or minutes is acceptable. For such applications processing can also be done offline or batch. In this case, the use of hardware acceleration is not necessary.

TABLE 1. Classification of edge computing applications by resource that determine most of latency and SoC component to be used for handling it.

Latency classification	Description	Application examples	Approach for handling latency
CPU-Bound	Latency due to intensive computations	Motor PID, Radar FFT	FPGA acceleration
RAM-Bound	Latency caused by memory accesses	EBig Data analytics on edge server	Optimized cache, DMA for FPGA
Bandwidth-Bounded	Latency related to data transfer	4K multi-camera video streaming	AXI-Stream + DMA to avoid congestion

However, the use of low-power CPUs (e.g. ARM Cortex-A9 that can be put in idle mode) can be appreciated to save energy. In particular, it can be appealing for the analysis of locally generated data logs, such as temperature and humidity data.

- Near Real-Time, with moderate latency: In this case, application response times of the order of 100 ms - 1 s are required. Tolerance for small delays is also acceptable, but not excessive so as not to hinder levels of modest interactivity. For example, the vehicle count in a given area for a traffic control system falls into this category. For these systems it is appreciable by the availability of CPUs with preemptive scheduling capabilities and the availability of FPGAs for pre-processing (e.g. resize and/or cross-encoding of images) is convenient for reducing the load on CPU cores, although not essential.
- Real Time, with critical latency: In this case, a further classification can be done based on the critical aspect that needs to be addressed. For these applications, an SoC equipped with an FPGA is extremely convenient because the FPGA can be used to accelerate algorithms for CPU-Bound applications, DMA handles transfers without CPU overhead and RAM-Bound applications can benefit from this, while highly specialized interfaces can be used for the transfer from the applications executed by CPUs and accelerated hardware.

Table 1 reports a further classification according to types of resources most responsible of latency issues. These classification categories and the SoC components that can be used to tackle the identified issues follow:

- CPU-Bounded: The FPGA accelerates algorithms (e.g. matrix multiplication algorithms for implementing AI technologies).
- RAM-Bounded: DMA handles transfers without CPU overhead.
- Bandwidth-Bounded: AXI-Stream can maximize throughput (e.g. for video transfer).

Finally, further classification based on the data generation process is necessary, as it is directly related to implementation strategies.

- Continuous Delivery. This class of applications generate a constant and predictable data flow (e.g., sensors at regular intervals for data acquisition from industrial machinery). In this case, the FPGA can be configured for fixed processing pipelines by making use of specialized interfaces, such as Advanced eXtensible Interface (AXI)-Stream, for seamless data transfers.
- Bursted Traffic. This class of applications generates and send data in intermittent bursts (e.g. data concerning rare but intense events, such as the recognition of anomalies and related inferences about them). In this case, the FPGA can be appropriately orchestrated, i.e. dynamically reprogrammed to handle processing peaks. In that case, the Block RAM (BRAM) in the SoC is an excellent choice for absorbing bursts and making data available for processing by FPGAs.

Clearly, an application can include multiple components that generate different categories of traffic. In this case each component can be designed considering its specific nature.

At this point, the problem is to identify implementation approaches capable of (i) make the best use of the technical features of the platform used, (ii) take advantage of a software deployment orchestration strategy in order to optimize the usage efficiency of the available SoC infrastructure.

To solve this problem, it is necessary to quantitatively characterize the available alternatives and make informed choices based on this characterization. This is the subject of the activities described below.

III. RELATED WORK

A. SoC-BASED EDGE COMPUTING SYSTEMS

The importance of FPGA-based hardware acceleration in edge computing was investigated in recent papers [30], [34]. In particular, in [36] the authors present EUKL, a Linux-based unikernel which runs in an RTOS user-space. The presented solution allows co-existence of Linux and RTOS applications within an embedded system while preserving RTOS real-time and reliability features. Moreover, the importance of SoCs in edge computing systems could be particularly advantageous, depending on the application requirements [18], [21], [29]. Key advantages includes a restricted space requirement, as a result of the combination of CPU, memory, I/O, and often hardware accelerators into one chip, energy efficiency, thanks to the optimization for low-power usage, making them suitable for battery-operated or remote edge applications, cost-effective, because they remove the necessity for individual components [40].

The results shown in this paper we present are based on a Zynq-7030 SoC, which belongs to the Zynq-7000 family. The SOC's in this family share the same processing core, but the 7030 offers relatively large and high-performance programmable logic, positioning it as a good choice for

embedded applications requiring substantial hardware acceleration. Furthermore, the methodological approach and most of the results shown in the paper are extendable to most popular devices similar to, or sometimes superior to, those of the Zynq 7000 family, as reported in Table 3.

Some developers using AMD devices make often use of PYNQ (Python Productivity for Zynq) [5], an open-source framework that simplifies programming FPGA-based SoCs devices using Python. For example, the paper [23] shows the implementation of a cluster, made energy efficient by using Zynq boards. The experimental results show the achievement of a gain factor in the range 1.75-85.5 on performance per watt, compared to a Ryzen server. In [20] the authors show a framework implementing the so-called FPGA GPU (FPGU) by using PYNQ for enabling the communication with a browser-based computing environment. A flexible and efficient FPGA-based architecture implementing the random forest algorithm is shown [13]. It turns out to be suitable for edge computing due to power limitations [22].

The expected advantages of PYNQ includes enabling FPGA programming using Python. This approach lowers the entry barrier for software developers without hardware expertise and simplifies offloading of critical functions to hardware. Developers also tend to resort to pre-configured bitstream files for common tasks as image processing. However, this tool brings also some disadvantages. Python introduces computational overhead compared to C/C++ or bare-metal FPGA designs and is not ideal for ultra-low-latency or high-efficiency applications. Moreover, pre-made overlays may lack flexibility for highly customized FPGA designs and advanced users still need the use of Vivado and Vitis design tools [6] and HDL knowledge for complex logic implementation. Moreover, Pynq is designed for Xilinx SoCs, as Zynq and Alveo, limiting portability to other FPGA vendors (e.g., Intel/Altera SoC FPGAs). Finally, not all versions of Vivado and Vitis are compatible with Pynq, including the most recent 2024 versions at the time of writing. Finally, even when compatibility is expected, introducing Pynq is far from straightforward and typically requires a long adaptation work [4].

For all these reasons, we decided to orchestrate the SoC using Linux bash commands for activating and deactivating the cores and hot-changing of the FPGA bitstream configuration file. The code for the FPGA was directly written in the Verilog language.

B. COEXISTENCE OF LINUX AND FreeRTOS

The coexistence of General Purpose Operating Systems (GPOS) and FreeRTOS in SoCs has become increasingly relevant, particularly for applications requiring both real-time performance and complex operating system features [41]. The GPOS considered in this paper is Linux, deployed by Yocto [42]. This way, the extensive library support of Linux can be combined with a lightweight real-time operating system designed for microcontrollers, focusing on delay performance

guarantees. Essentially, Linux can manage high-level tasks, while FreeRTOS can handle time-critical operations.

Some papers [14], [31], [33] put forward the concept of ARM trust zone, in order to deploy an RT execution environment alongside Linux. A traditional RTOS (tRTOS) is used to execute the secure functions, while a GPOS the other functions regarded as non-secure.

More recently, this coexistence was leveraged in [41], where the authors show implementation solutions by using a Raspberry Pi 4B board. In particular, they build the so-called compounded RT operating systems (cRTOSs), where a GPOS and a traditional RTOS run on two virtual machines executed by a hypervisor.

An interesting proposal for implementing a robot controller is shown in [38]. This proposal consists of a dual-core GPOS-RTOS dual operating system controller. The authors do not show implementation details, but show experimental validation of real-time performance.

The GPOS-FreeRTOS combination is also useful for space applications. The paper [17] shows the combination of the so-called NASA Operating System Abstraction Layer for FreeRTOS running on an Arm Cortex-M microprocessor.

In [2] the authors analyze the framework OpenAMP as the basis of the coexistence of Linux and FreeRTOS using the Zynq UltraScale+ MPSoC ZCU102 evaluation board. The hardware design, shown in the paper, leverages the AXI-GPIO interface connecting the PS and the PL of the SoC. It is thus not surprising that the authors noticed some significant delays for executing some tasks characterized by real-time constraints. In this paper we show a detailed comparison in the use of AXI-GPIO and AXI-Stream interfaces to highlight their suitability for implementing different functions.

In [15], the authors propose a Real-Time RTOS-GPOS Remote Procedure Call for a dual-OS system. Time predictable control is achieved through three mechanisms: Software Generate Interrupt message transforming, interrupt handler RPC serving, and priority-swapping. After an extensive comparison of the alternatives, they observe a number of issues in developing an RPC service in a hybrid dual-OS system.

It is also worth mentioning the doctoral thesis [32] and the master's thesis [11] that include an experimental validation of Asymmetric Multi Processing in dual-core ARM Cortex-A9 processor, in particular for memory access and PS-PL interfacing.

IV. BACKGROUND

A. ZYNQ 7000 SOC

The Zynq-7000 series, developed by Xilinx, is a SoC technology that integrates a dual core ARM-based processing systems (PS) with a programming logic (PL) including an FPGAs [8]. This combination allows the creation of flexible platform suitable for hosting a wide range of applications, such as ML inference, video processing, software-defined radio. The internal SoC architecture enables developers

TABLE 2. Comparative table of SoC FPGAs similar to Zynq-7000.

SoC Family	Manufacturer	PS (Processor)	PL (FPGA)	External Memory Support	Overall Comparison
Zynq 7000 [8]	AMD (Xilinx)	Dual-core Cortex-A9	Artix-7 / Kintex-7	DDR3, DDR3L, DDR2, LPDDR2	Baseline
Zynq UltraScale+ [39]	AMD (Xilinx)	Quad-core Cortex-A53	UltraScale	DDR3, DDR4, DDR3L, LPDDR3, LPDDR4	Superior
Intel Cyclone V [24]	Intel (Altera)	Dual-core Cortex-A9	Cyclone V	DDR2, DDR3, LPDDR2	Comparable
Intel Arria V [3]	Intel (Altera)	Dual-core Cortex-A9	Arria V	DDR3, QSPI, EPCQ, EEPROM	Superior (PL)
AMD Versal [9]	AMD	Cortex-A72 + AI Engines	Adaptive Compute	DDR4, LPDDR4/4X, RLDram3, QDR-IV	Superior

implementing customized hardware and software solutions, making it suitable in the evolving landscape of edge computing systems. This versatility supports the development of applications across various domains, including industrial automation, telecommunications, and the Internet of Things (IoT). Furthermore, the platform compatibility with different operating systems, such as Linux and FreeRTOS, facilitates integration into other existing systems.

A block diagram of the PS portion of a Xilinx Zynq System-on-Chip (SoC), which combines ARM processors with programmable logic is shown in Figure 1. The key components shown in the figure are

- The I/O Subsystem, that includes the Bank0 MIO (Multiplexed I/O) pins (15:0), the Bank1 MIO pins (53:16), various I/O peripherals (SPI, I2C, CAN, UART, etc.) the Flash memory interfaces including Quad SPI and the SMC (Static Memory Controller) timing calculation.
- The Central Processing System, including a Dual ARM Cortex-A9 CPUs with L2 cache, On-Chip Memory (OCM) with 256KB SRAM, DDR memory controller, and Various interconnect buses.
- The System Services, including Clock generation and reset control, DMA channels, Security modules (AES/SHA), and Debug components (CoreSight, DAP).
- The Programmable Logic Interface, with Extended MIO (EMIO) for connecting to PL, PS-PL clock ports, and AXI interfaces for high-bandwidth communication.

Software implementation on the Zynq-7000 family poses several challenges, that particularly concern the mutual integration of the available hardware components of the SoC and the software components that regulates their behavior, including debugging and dynamic deployment.

Developers must typically navigate these hurdles to fully harness the SoC capabilities, highlighting the need for a comprehensive understanding of both hardware and software design principles. The complexity of the Zynq-7000 architecture, along with the rapid pace of technological advancements, has spurred ongoing discussions within the developer community regarding best practices and methodologies for effective deployment [19], [35], [37].

For the hardware configuration and programming of the SoC, the main tools designed for this purpose were used, namely Vivado and Vitis [6]. Both are tools developed by Xilinx (now AMD) for the design and development of FPGA and SoC (System-on-Chip) based systems.

Vivado is an integrated design environment (IDE) that produces the synthesis of the hardware design by translating HDL code (VHDL/Verilog/SystemVerilog) into a configuration file for the FPGA. It produces configurations optimized for speed, and power consumption. It also supports pre-synthesis and post-synthesis simulation to verify design behavior and advanced debugging using an Integrated Logic Analyzer (ILA) and VIO (Virtual Input/Output) for real-time monitoring and control of signals. Vivado also includes a library of pre-configured IP (Intellectual Property) blocks. An IP block is a pre-designed and reusable hardware module, which implements a specific functionality, such as a processor, a USB interface, and a memory space. Vivado also includes the possibility to create custom IPs by the so-called IP Integrator tool. Regarding the SoC used in this work, the IP of the Processing System of the Zynq 7000 family is shown in Figure 1.

Vivado is typically used in conjunction with Vitis, a development platform for accelerated applications on FPGAs and SoCs, which combines software for the CPU and hardware configuration of FPGAs. Vitis supports C/C++/OpenCL to develop hardware-accelerated kernels and allows integration with Vivado HLS (High-Level Synthesis) to convert C/C++ code to RTL. Therefore, using Vitis it is possible to realize joint development of firmware (PL) and software (PS) for SoCs, such as the Zynq-7030 used in this work. We should also highlight the support for bare-metal, RTOS and Linux, which have been used for the experimental implementation.

B. ASYMMETRIC MULTIPROCESSING AND OpenAMP

Multiprocessing is a technique that enables a computer system to utilize multiple processors to improve performance and reliability. There are two categories of multiprocessing, Asymmetric Multiprocessing (AMP) and Symmetric Multiprocessing (SMP). The key distinction between them lies in how tasks are distributed in a coordinated manner among

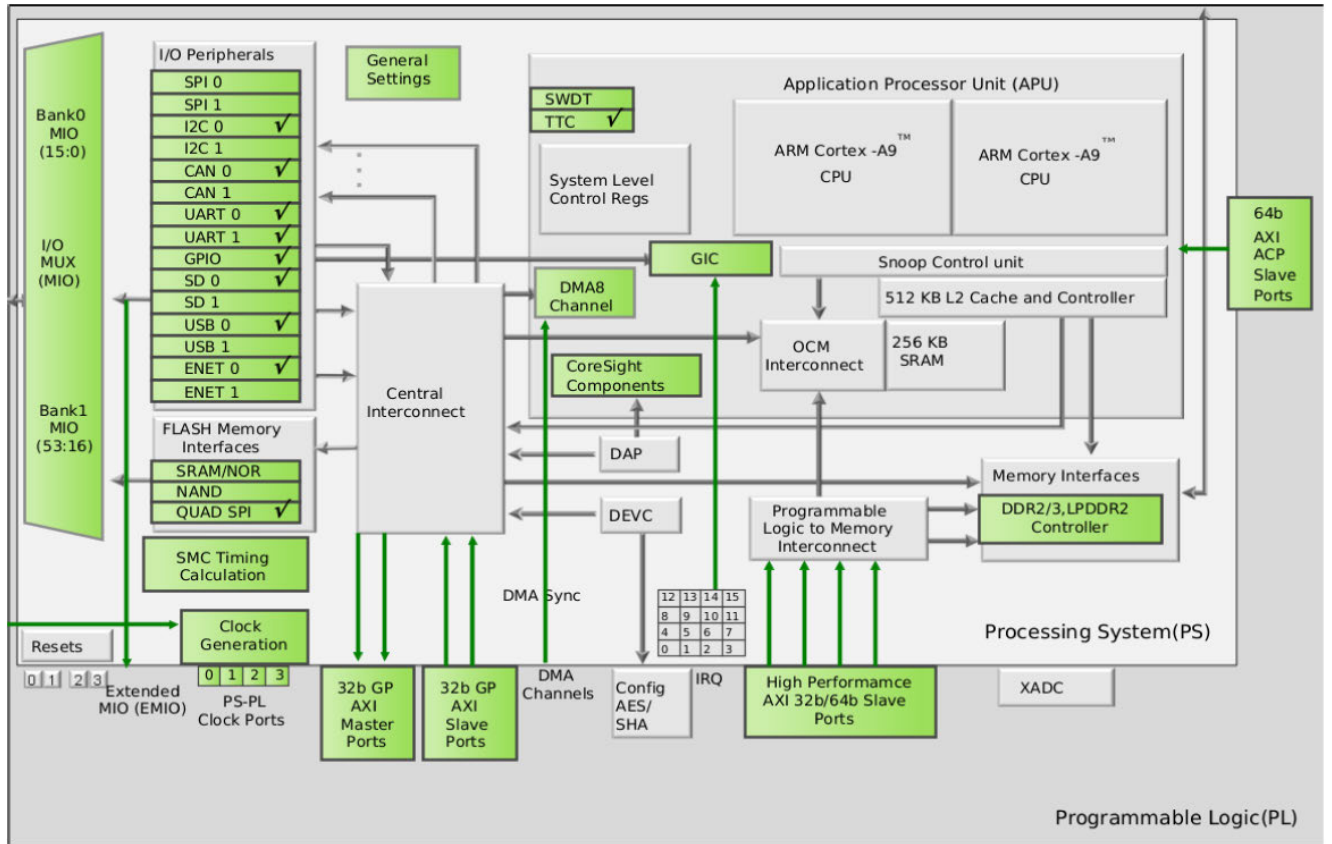


FIGURE 1. Vivado IP of the programmable logic of the Zynq 7000 SoC.

processors and how they communicate. In AMP, one processor, referred to as the master processor, oversees the other processors, which assume the role of slave processors. In this setup, all operating system functions and task scheduling are managed by the master processor. Differently, in SMP all processors have equal access to the system and memory resources. Each processor operates independently while performing tasks and interacting with the operating system.

In this work we used AMP to achieve process isolation and guaranteed real-time. In fact, in AMP, FreeRTOS has exclusive control over one core, avoiding interference from Linux, that is not real-time. Furthermore, the Zynq-7000 family and similar platforms provide efficient mechanisms for inter-process communication (IPC) in AMP, such as Shared Memory (DDR or On-Chip Memory - OCM) for data exchange and interrupts for notifications (e.g. from FreeRTOS to Linux). These mechanisms are easier to manage in AMP than in SMP, where synchronization would require a unified kernel. In particular, we utilized the OpenAMP (Open Asymmetric Multi-Processing) open-source framework [27]. It includes the software components required to implement software applications for AMP systems. By using this framework, different architectures that permits AMP applications to take advantage of the multicore intrinsic parallelism can be deployed.

OpenAMP is composed of several libraries and modules:

- Libmetal, which provides a low-level API for accessing hardware (memory, interrupts, devices).
- RPMsg (Remote Processor Messaging), which is a communication protocol based on message passing. It uses shared memory and interrupts for message exchange between cores and supports virtual channels (*VirtIO*) for communication between operating systems, described below. OpenAMP implements RPMsg via shared memory utilizing a ring buffer. It makes use of *VirtIO* devices as abstraction layer at system level [26]. Fig. 2 illustrates that the entire communication implementation, that can be divided into three distinct layers - Transport, Media Access Control, and Physical layer. Each of these can be implemented independently.
 - Physical Layer: The basic setup requires a single interrupt line for each communicating core. It should be noted that there is no requirement for inter-core synchronization hardware elements like inter-core semaphore or queues. This is due to the behavior of the *virtqueue* process, which employs single-writer-single-reader circular buffering, described in the subsequent layer. The utilization of interrupts by the ring buffers is not mandatory and can be disabled. Nevertheless, both cores must query these

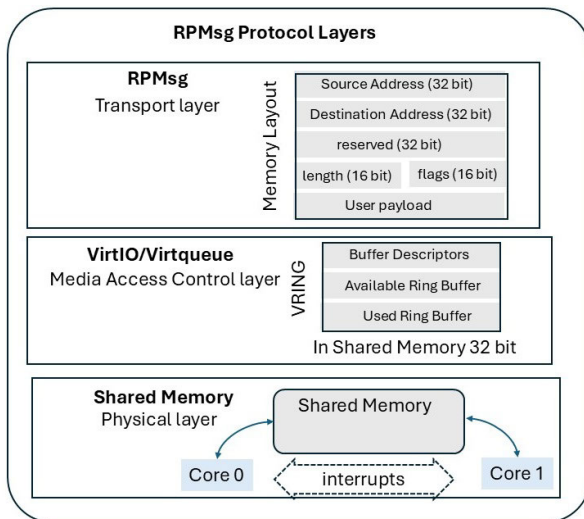


FIGURE 2. RPMsg protocol layers: The OpenAMP RPMsg functions can be separated in three different layers, namely transport, media access control and physical layer.

ring buffers for any new incoming messages, which might not be the most efficient approach.

- **Media Access Layer - VirtIO:** This layer is a crucial component of the architecture, since it eliminates the requirement for inter-core synchronization. This is accomplished through a method known as single-writer single-reader circular buffering, called Vring, a data structure that allows multiple asynchronous contexts to exchange data. The Vring includes three components, namely the buffer descriptor pool, the available ring buffer, and the used ring buffer. All of them are located in the shared memory. Every buffer descriptor includes a 64-bit buffer address, its length represented as a 32-bit variable, a 16-bit flags field, and a 16-bit link to the subsequent buffer descriptor.
- **RPMsg Header Definition:** An RPMsg message is included within a buffer in the shared memory, pointed by the address field stored in the Vring buffer descriptor. The initial 16 bytes of this buffer are utilized internally by the RPMsg Transport layer. The first 32 bits represent the address of the source endpoint, while the following 32 bits indicates the receiver endpoint. A reserved field is present just for 16-bytes alignment purposes. The following 16 bits specify the length of the payload and other 16-bit flags field ends the header. The user payload is placed after the RPMsg header [28].

- **OpenAMP Lifecycle Management,** which manages the start/stop of remote cores (e.g. how to start FreeRTOS on Core 1 while Linux runs on Core 0).
- **Proxies & Remoting,** which allows to call remote functions (e.g. a task in FreeRTOS can invoke a function in Linux and vice versa).

V. SOLUTIONS FOR DATA EXCHANGE

The flexibility of the Zynq 7000 SoC allowed us to explore different implementation alternatives for designing services. Given the emphasis placed in this paper on the real-time nature of applications, it is necessary to have a complete picture of the communication capability between the components of the SoC.

For the sake of completeness, we considered applications based on the coexistence of different operating systems in the SoC and the parallel availability of hardware acceleration. For this reason, before showing the implementation solutions it is necessary to characterize the possibilities of interaction between the PS cores and between PS and PL. In particular, the characterization is carried out considering the achievable latency and throughput, depending on the size of the exchanged packets.

For this characterization, the following test scenarios are analyzed.

- LINUX ↔ FreeRTOS
- LINUX ↔ FPGA
- FreeRTOS ↔ FPGA

A. DATA EXCHANGE BETWEEN LINUX AND FreeRTOS

This section illustrates the operating procedure starting from the hardware design to the implementation of the application software distributed on the two cores of the Cortex A9 processor. This procedure includes resource partitioning (memory and peripherals), booting of secondary cores from Linux, loading FreeRTOS on the secondary core, data exchange between Linux and the secondary core through RPMsg, and concurrency management.

The hardware project for this first analysis is shown in Figure 3. The project essentially includes the PS Zynq 7000 shown in Figure 1 and the minimum peripherals for interacting with the external.

Through the hardware project, a file in xsa (Xilinx Support Archive) format was generated using Vivado. It includes basic information on the project hardware configuration, along with address maps for the PS and driver information. This xsa file was used by the software developed to interact with the hardware environment, identify the system peripherals and the memory address map.

The software architecture is based on the use of interrupts and shared memory. The Libmetal library includes APIs that can be used to access devices, handle device interrupts, and request memory in different operating environments. Using Libmetal, it is possible to create an AMP solution for packet exchange. In particular, the used OpenAMP library provides a framework for remote processor management and inter-processor communication using key lifecycle management and inter-processor communication features for managing remote compute resources. Furthermore, it provides a standalone library for use with FreeRTOS and baremetal software environments, and compatibility with upstream Linux *remoteproc*, *rpmmsg*, and *VirtIO* components.

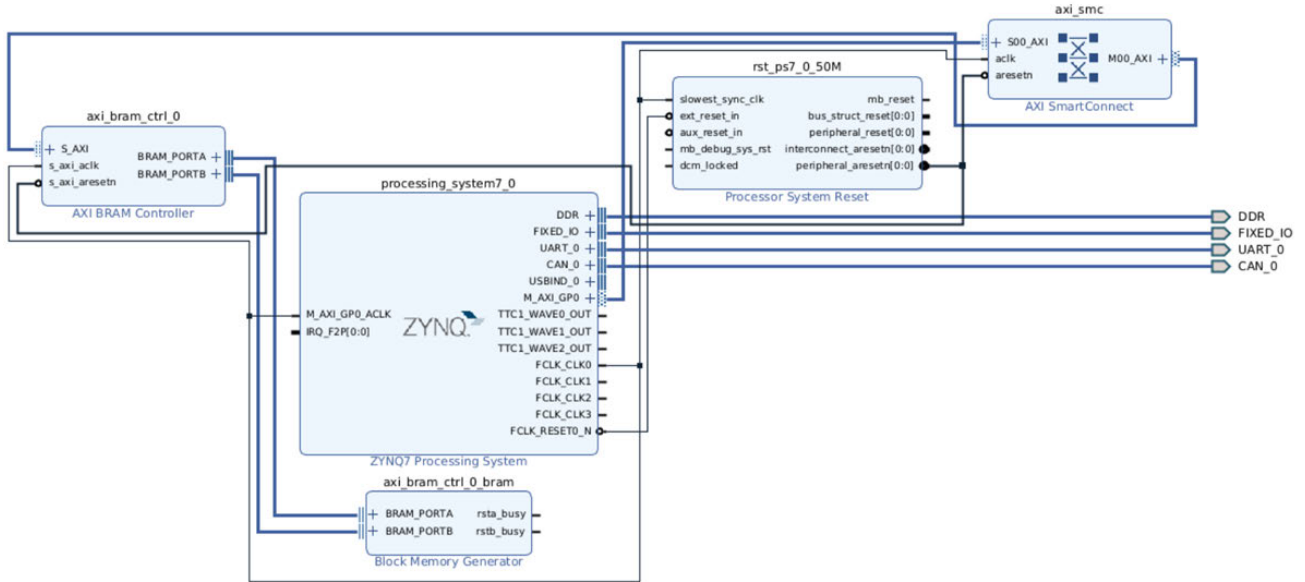


FIGURE 3. Hardware design for analyzing interactions between cores running Linux and FreeRTOS.

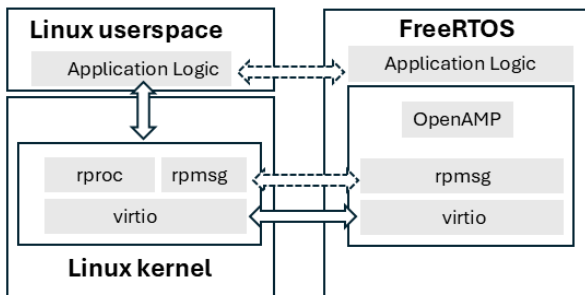


FIGURE 4. Communication model between Linux and FreeRTOS. Dashed arrows indicate logical relationships.

The latter point is fundamental because it represents the basis of exchange between Linux and FreeRTOS operating systems. In particular, the OpenAMP *VirtIO* library implements the *VirtIO* standard for shared memory management. *VirtIO* is a virtualization standard for network and disk device drivers in which only the driver on the guest device is aware of running in a virtual environment, with the hypervisor. *remoteproc* provides the functionality for managing the lifecycle of remote processors. This library, using the information published via the remote processor firmware resource table, allocates system resources and creates *VirtIO* devices. The RPMsg API allows IPC between processes running on independent cores in AMP. The resulting architecture is sketched in Figure 4.

The basic steps of the implementation on the Linux side, which acts as the logic controller for setting up the OpenAMP channel for message exchange, is shown in Figure 5. The following steps are executed:

- Definition of a firmware resource table. The resource table defines the firmware entries required for the

OpenAMP application. This is a list of the system resources required by *remoteproc*.

- Creation of the *remoteproc* structure using the resource table.
- Definition of RPMsg callback functions.
- Creating the RPMsg *VirtIO* device.
- Creation of an RPMsg endpoint and association of the RPMsg device with callback functions.
- Usage of the system call *write()* by Linux to send a message and the *rpmsg_send()* API by FreeRTOS to send the echo back to the remote processor.

After the environment is initialized, the flow of the OpenAMP application consists of the RPMsg channel which acts as communication between the master and the remote processor via the RPMsg I/O and *send()* callback functions.

The steps illustrated have been implemented in a programs written in C language. Although the detailed description of the implementation of the long and complex C programs used for this analysis is much beyond the scope of this paper, we both provide some details in order to give interested readers a view of the basic steps of the implementation and in we have set up a URL from which the programs used in the analyzed case studies can be downloaded [1].

The Linux program acts as a client, while the FreeRTOS program acts as a server, which has to be previously loaded by Linux. The program has been implemented by integrating the Vitis template *echo-test* [7]. The essential steps related to sending packets from Linux to FreeRTOS are depicted in the flow diagram shown in Figure 6. The application begins execution in user space on the Linux processor. It dynamically loads the *rpmsg_char* kernel module, ensuring the availability of the character device interface required for RPMsg

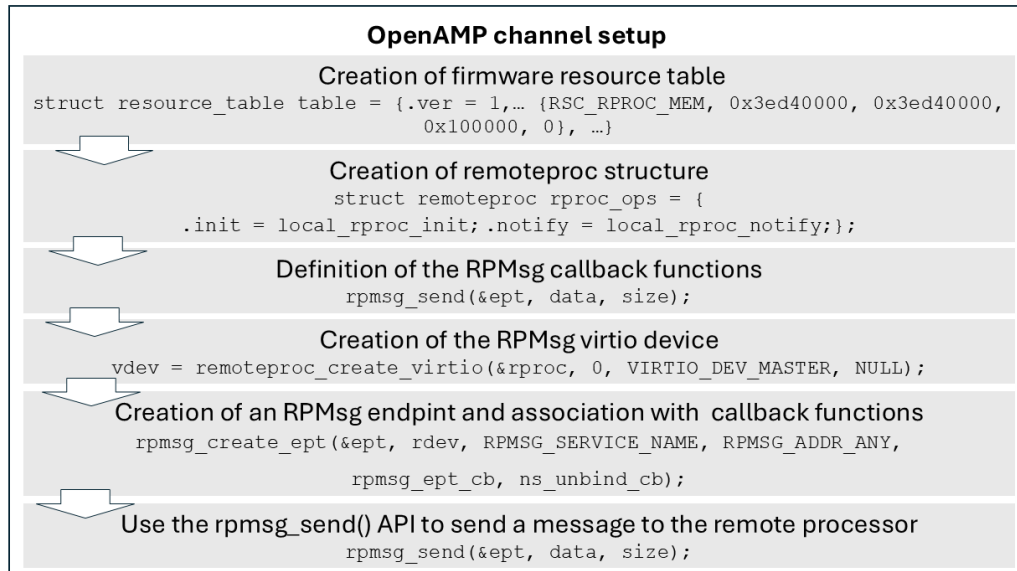


FIGURE 5. Steps for setting up the OpenAMP channel between Linux and FreeRTOS with sample C functions.

communication. It then verifies the presence of the specified RPMsg device within the system *sysfs* hierarchy. To route communication through the *rpmsg_char* interface rather than a default handler, the application explicitly binds the RPMsg device to the character driver using the *driver_override* mechanism, followed by a manual write to the *bind* file in *sysfs*. Once the device is bound, the application opens the character device associated with the RPMsg control channel. It then creates an endpoint via *ioctl()* call on the character device. This endpoint is used to establish a logical connection with a corresponding endpoint on the FreeRTOS side. Next, memory is allocated for transmit and receive payload structures. The core loop of the application iterates over the configured number of test rounds. Within each round, the time is calculated with timestamps captured at the beginning and end of the inner loop. The nested loop performs the actual data exchange, sending a predefined number of payloads. Each payload is initialized with a repeating generic pattern (0xA5 in our experiments) and transmitted using the *write()* system call. After each transmission, the application enters a read loop to receive the echoed response. Upon receiving the response, the payload integrity is validated by checking each byte of the data buffer for any corruption. Once all payloads have been processed, the application calculates and prints the total elapsed time. Finally, the application frees all allocated memory and cleanly closes the file descriptors. The program then exits, returning control to the shell.

The part of the logic executed by FreeRTOS corresponds to that of the *echo-test* application [7], which, for the latency measurement, did not need to be modified.

The application used for evaluating performance consists of executing 1000 iterations, each with a minimum packet size of 1 byte, plus another 8 bytes due to additional fields inserted

in the Linux-side payload, 4 bytes to identify the packet with a sequence number, and another 4 for the packet size itself. This is because the user-space Linux application cannot access the RPMsg protocol headers, which are managed at the kernel level, so part of the payload is used to create an additional application header.

B. DATA EXCHANGE BETWEEN LINUX AND FPGA

This section describes the procedure implemented in the application used to evaluate the data transmission bandwidth between the Linux system in the PS and the FPGA present in the SoC. In this specific case, the FPGA was configured to perform a simple NOT logic operation, in order to verify the correct functioning of the system without requiring significant processing time capable of significantly altering the estimates. In this way, the measured time is that related to the data transfer from the Linux system to the FPGA and the transmission of the processed data in the opposite direction. In the Xilinx Zynq-7000 SoC series, the Processing System, which includes the dual-core ARM Cortex-A9, can communicate with the PL (including the FPGA fabric) through different interfaces, which differ in complexity of use and performance. In this analysis we focused on the two extremal solutions, relate to the GPIO (General Purpose Input/Output) Interface and the AXI4-Stream. The former is designed for general purposes, i.e. for controlling and reading simple digital signals between the PS (ARM cores) and the PL (FPGA) or external devices. It is easier to configure and requires low resource usage. For this reason it is considered suitable for basic I/O tasks. Differently, the AXI4-Stream interface is hardware-accelerated and supports efficient transfer of bulk data. Although the mentioned features can be guidelines for application design, an experimental analysis is necessary to rigorously frame the

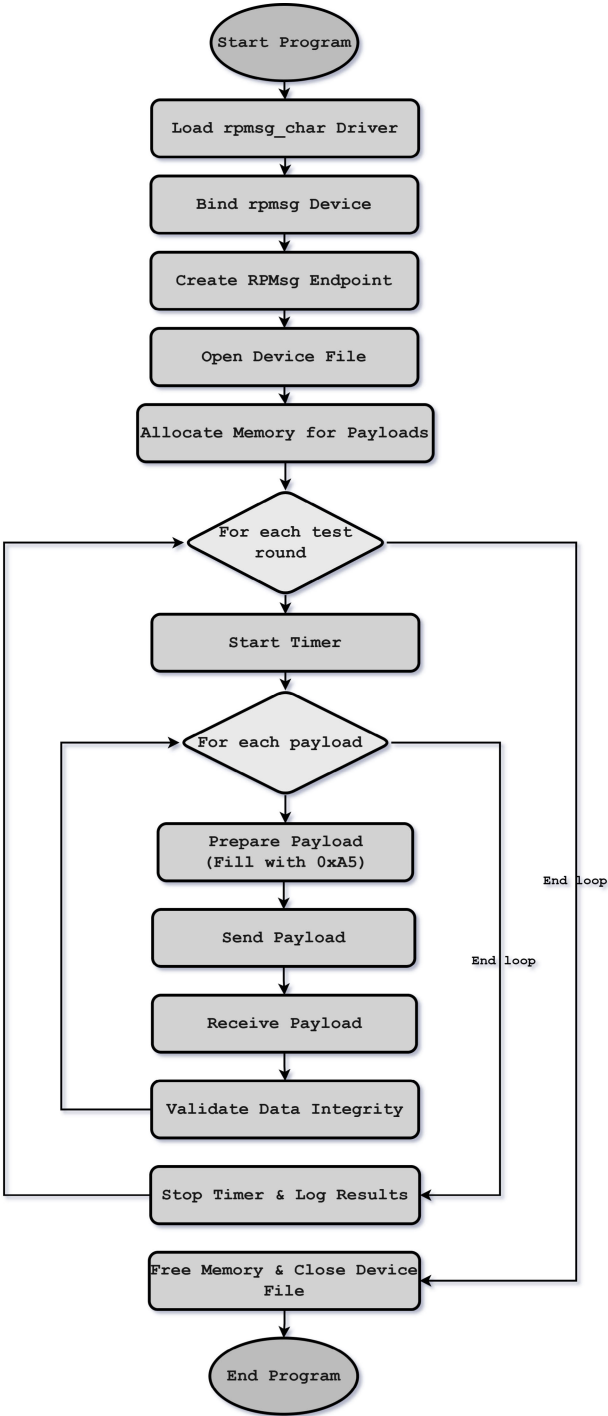


FIGURE 6. Linux-side flow diagram of the C program used for latency and bandwidth estimation between cores.

possibilities offered by the two alternatives and make the optimal choice based on the application characteristics.

In both cases, a hardware project must be prepared to use the various interfaces and associated modules.

The hardware design for using the GPIO is shown in Figure 7. As shown in the figure, a custom FPGA block implementing of the NOT function is used. The PIs related

to the GPIO ports are used to send data to the FPGA and to receive data from the PI of the PS of the Zynq-7030 SoC. The other PIs in the figure have only a physical interconnection role to the bus.

A C language program was written to send a 32-bit hexadecimal value to the FPGA for a total of 1000 iterations, in order to calculate the average throughput and latency. Figure 8 shows the flow diagram of the program, such as memory management for the GPIO interfaces and data exchange. The program implements a bidirectional communication loop between a Linux-based processor and the FPGA using AXI GPIO peripherals. It begins by accessing physical memory through `/dev/mem`, mapping the AXI GPIO regions responsible for both input and output channels. Once mapped, the GPIO directions are configured: The output channel is set to transmit data to the FPGA, while the input channel is configured to receive responses. The execution timing is measured by capturing timestamps at the beginning and end of the transmission loop. The core processing loop iterates a specified number of times, transmitting a fixed bit pattern via output GPIO and immediately reading back the corresponding data from the input GPIO. This process emulates a simple hardware handshaking mechanism. After all iterations, the total elapsed time is computed and reported. The program concludes by unmapping the memory regions and closing the device, ensuring proper resource cleanup.

Regarding the use of the AXI4-Stream interface, it is necessary to access the High Performance (HP) ports of the SoC, to obtain high-speed data transmission between the PL and the operating system. The use of the HP ports, designed to support high-bandwidth communications between the PL and the system DDR memory, allows the FPGA to directly access the system memory via a DMA (Direct Memory Access) controller. In the specific case of the hardware implementation for this application, two HP 64-bit interfaces were enabled, referred to as HP0 and HP1.

Next, it was necessary to integrate two instances of the AXI DMA block into the Block Design in Vivado. These two blocks play a fundamental role in establishing a high-performance bidirectional communication channel between Linux and the FPGA, leveraging the AXI4-Stream interface and the newly enabled HP ports. In particular:

- The first AXI DMA block was configured to receive data from the Linux system through the HP0 port, transferring it from the DDR memory to the FPGA logic block via the MM2S (Memory-Mapped to Stream) channel.
- The second AXI DMA block, instead, was set up to send the data processed by the FPGA to Linux, using the HP1 port and the S2MM (Stream to Memory-Mapped) channel, which allows writing directly to the system DDR memory.

Both AXI DMA blocks are configured to support 64 bits data transfer. To enable the connection between the HP0 and HP1 ports of the PS and the AXI DMA modules, AXI SmartConnect modules have been included in the hardware

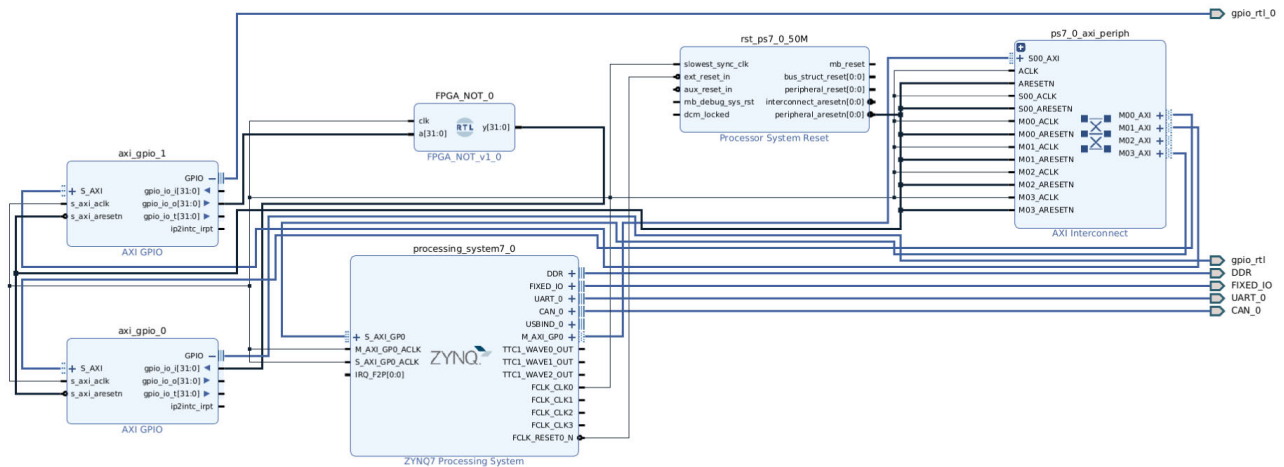


FIGURE 7. Hardware design based on the use of GPIO interfaces.

design. The resulting hardware configuration is shown in Figure 9.

The *Not_TEST_AXI* module is designed to receive data through an AXI4-Stream interface and apply a logical NOT operation on the received data. The module has an architecture based on two main AXI4-Stream communication channels, one input and one output. The input interface (AXI4-Stream Slave) receives data from the system, while the output interface (AXI4-Stream Master) sends the processed data.

To develop the C code for data exchange, it was necessary to program the AXI Stream protocol control signals to regulate and synchronize the data transmission between the master and the slave. Specifically, when data is received by the module, it is read via the *s_axis_tdata* signal. The module checks whether the input data is valid (via the *s_axis_tvalid* signal) and whether the output is ready to receive it (via the *m_axis_tready* signal). When both of these conditions are met, the module performs the logical NOT operation on the received data and stores the result in the *m_axis_tdata* signal, which represents the data to be sent out. Once the data has been processed, the module ensures that it can be transmitted correctly. The *m_axis_tvalid* signal is set to 1 to indicate that the processed data is ready to be sent. However, the module will not send the data if the target system is not ready to receive them, as indicated by the *m_axis_tready* signal. If the target system is ready, the data is sent, and the *m_axis_tlast* signal is used to signal that the sent data packet is the last one in the flow. The *s_axis_tready* signal is set to indicate that the module is ready to receive new data only when the output is ready to send them. This avoids conflicts between receiving and transmitting data, ensuring that the module does not receive new packets until it has finished sending the previous ones.

Figure 10 shows the flow diagram of the code regulating data transfer, highlighting the principal phases of the logic implemented. The program implements a high-throughput data transfer routine using AXI DMA. It opens the */dev/mem* device to enable direct access to physical memory regions,

mapping both MM2S and S2MM DMA control register as well as a contiguous DDR memory segment used as the transmission and reception buffers. Following memory mapping, the transmit buffer is initialized with a predefined data pattern, while the receive buffer is cleared to zero. Both DMA channels are reset, configured with appropriate source and destination addresses and then enabled for operation. The main processing loop iterates over a user-defined number of payloads, updating the transmit buffer at each iteration and launching DMA transfers by writing to the appropriate length registers. Completion is monitored through polling of the DMA status registers. Upon completing all transfers, the receive buffer is validated against the expected data to ensure data integrity. Execution time is measured by capturing timestamps at the beginning and end of the transfer loop. Finally, the program outputs performance metrics and releases all allocated resources.

C. DATA EXCHANGE BETWEEN FreeRTOS AND FPGA

As for measuring the performance in the communication between FreeRTOS and the FPGA, again implementing an error checking mechanism, both the hardware model and the Verilog code used for programming the FPGA shown above have been reused. However, for what concerns the C code, some modifications were required in terms of programming logic, libraries, and hardware interaction. FreeRTOS operates in a bare-metal model, in which static allocation is preferred and memory coherence must be controlled manually when the FPGA makes use of the shared memory.

Whilst Linux abstracts hardware using *sysfs* files for GPIO and kernel drivers for DMA, FreeRTOS requires programming registers. For example, the *XGpio* driver is not a kernel module but a library and its functions are called directly. To read a GPIO it is necessary to invoke *XGpio_DiscreteRead*, which pokes the register itself. DMA is managed through a sequence of register writes, orchestrated via Xilinx *XAXiDma* library. Also the interrupt managements becomes more complex, since interrupts cannot be triggered by kernel-managed handlers and

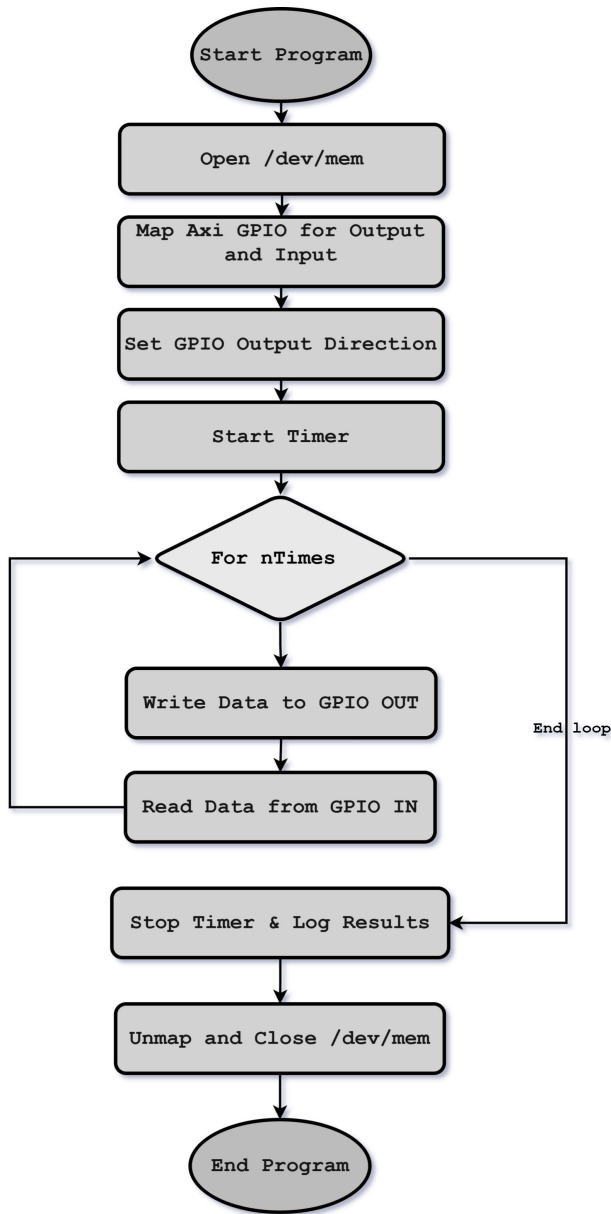


FIGURE 8. Flow diagram of the Linux-side C program used for GPIO interfacing and data exchange with the FPGA.

their management is deferred via semaphores or task notifications. Time and scheduling management is also different. FreeRTOS treats time as a quantifiable resource that must be directly managed.

For the experimental analysis, two sub-case studies can be identified for the data exchange between PS and PL, namely the one related to the use of the GPIO interface and the one related to the use of AXI4-Stream.

The exchange between FreeRTOS and FPGA via GPIO is a logically similar approach to that of communication between Linux and the FPGA. The communication time between FreeRTOS and the FPGA was measured using the *xTaskGetTickCount()* function, which returns the number

of system ticks that have passed since startup. The value obtained is then divided by the frequency of system ticks, thus allowing the time to be expressed in seconds. Since FreeRTOS is a real-time operating system, time management occurs through a tick timer, which is a periodic interrupt generated by a hardware timer. The timestamp thus obtained was recorded in the instant immediately preceding the sending of the first value to the FPGA and in the instant immediately following the sending of the last value.

In the case of using the GPIO interface, at startup the application initializes three main peripherals:

- UART (XUartPs): used for sending log messages and for printing the test results on the terminal.
- GPIO TX (XGpio): configured as an output channel, used to transmit data from the processing unit to the FPGA.
- GPIO RX (XGpio): configured as an input channel, used to receive the data returned by the PL.

On the FreeRTOS side, the main component of the application is the *xTaskCreate* library function, which is used to run a loop for a predefined number of iterations, corresponding to the number of exchanged packets, which is 1000 for this analysis.

During this loop the following steps are executed:

- 1) An incremental value *txCount* is transmitted via the outgoing GPIO.
- 2) The value is read from the incoming GPIO channel.
- 3) The received value is checked to be the bitwise complement of the sent value.
- 4) The counters for received packets and any comparison errors are updated.

For evaluating performance, the *xtime_l.h* module is used, which provides the elapsed time in ticks. The start and end time of the test is recorded via *XTime_GetTime*, and the throughput is calculated by dividing the total volume of data transmitted by the total time, thus obtaining the transmission throughput.

In the case of using the AXI-Stream interface, the main peripherals initialized at application startup are:

- UART: used to send log messages and test results to the terminal;
- DMA TX and RX: used to transmit and receive data between the main memory and the FPGA through an AXI-Stream interface.

The memory regions used for the transmission and reception buffers are predefined via macros (*TX_BUFFER_BASE* and *RX_BUFFER_BASE*) and must reside in a portion of memory accessible by both the processor and the DMA. This portion can be checked within the project in Vivado. The main task of the application performs a transfer cycle repeated for a defined number of iterations, 1000 for this analysis.

In each iteration:

- 1) The data in the transmission buffer is updated;

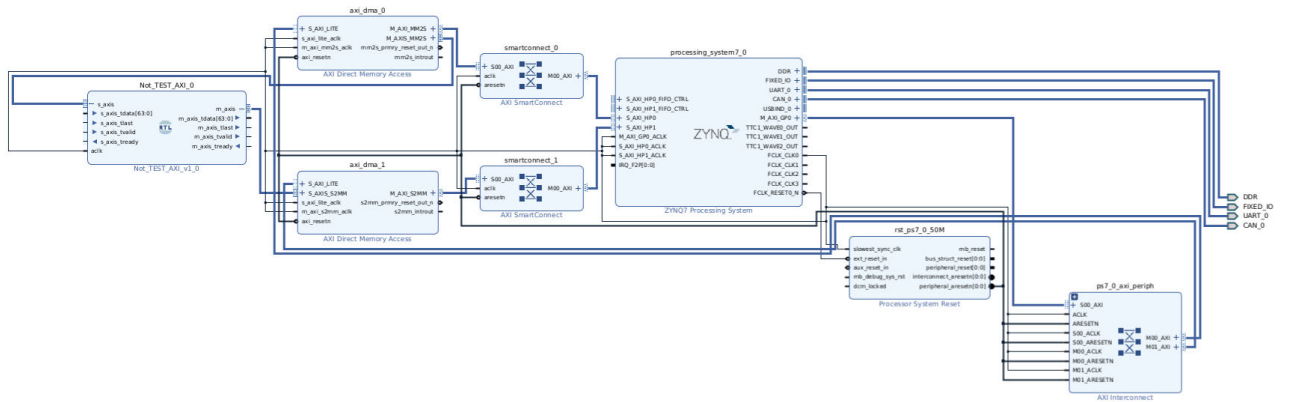


FIGURE 9. Hardware design based on the use of AXI-Stream interfaces.

- 2) The cache is managed to ensure consistency between memory and peripherals (flush for the TX buffer, invalidate for the RX buffer);
- 3) The RX transfer is started, followed by the TX transfer;
- 4) The system waits for both DMA operations to complete;

The overall execution time is measured using the FreeRTOS tick counter, from which the throughput is calculated.

D. CONFIGURATIONS DETAILS

For what concerns the detailed configuration of the experimental platform, we used the default system configuration (e.g., in PS main clock value and relevant peripherals, see also [10]), unless changes were strictly necessary. For instance, we did not use CPU pinning and isolation, except in experiments related to the communication between Linux and FreeRTOS, where both CPU cores was initially allocated to Linux, and then CPU core1 was dynamically allocated to FreeRTOS at runtime, leaving the Linux system with just CPU core0. In this regard, the complete configuration of the RPMsg protocol is shown in documents available in the supporting material [1]. For instance, we used `vring0` for the communications from Linux to FreeRTOS, and `vring1` for the reverse direction. Their size is 16 KB, whereas the pool for the RPMsg buffer is 1 MB. Finally, the memory reserved for the whole firmware is 16 MB. We used interrupts for synchronization operations. The policies for the usage of the DMA can be found in the Vivado logs [1]. Finally, we used the suggested values for compiler flags, such as those reported in OpenAMP documentation [7].

When carrying out measurements, we set a number of initial warm-up iterations equal to 50, then we collected latency measurements for the following 1000 iterations. We recorded the timestamp immediately before and after each communication exchange. As for the time resolution, in Linux we used the `CLOCK_MONOTONIC_RAW`, equal to 1 ns, and in FreeRTOS we used a high-resolution timer based on a frequency of 666 MHz, thus with a resolution of 1.5 ns. In fact, the base timer (tick) is equal to 10 ms (100 Hz) and thus not suitable for the accuracy needed in the measurement campaign.

VI. EXPERIMENTAL RESULTS OF TEST SCENARIOS

The experimental results on latency and throughput between Linux, FreeRTOS, and the FPGA in the Zynq-7000 is critical for edge computing applications to fully take advantage of the SoC capabilities.

In Table 3 we report the latency values evaluated through experiments. It is important to note that the minimum message payload used to evaluate the minimum latency also minimizes the throughput, confirming the known trade-off. Higher throughput values are achievable by using larger payloads. In what follows we will delve deeper into the analysis of the throughput.

The performance differences are not due to a single factor, but rather the combination of the communication path and the mechanism used. These factors determine the overhead at every level of the stack.

As for the latency measurement between Linux and FreeRTOS, these are by far the slowest paths by a significant margin. A minimum value of 1 byte of the RPMsg payload was used, to include the minimum transit time in the endpoints in the measurement. Given the experimental setup, the measurements were quite stable. Therefore, it was possible to estimate the average latency of the round trip time of the inter-core Linux $\rightarrow$ FreeRTOS $\rightarrow$ Linux communication in about 69 μ s. These values are due to the fact that the data exchange between different cores makes use of a protocol RPMsg designed for control and low-bandwidth data, not high-speed streaming. OpenAMP is a framework for managing inter-processor communication, using shared memory. The protocol stack, including packetization and error-checking, adds significant overhead. We point out that these latency measurements required an extended warm-up procedure compared to all the other experiments carried out in our test campaign. Quantitative details of the measurement setup are reported in Figure 11 and Figure 12. As shown in the first figure, both the average latency value and the corresponding standard deviation become stable only when the fraction of warm-up samples is at least 20%, i.e. 200 samples. For smaller values, corresponding to the typical configuration

TABLE 3. Throughput and latency in Zynq-7000 communication paths via shared memory: average values are provided with standard deviation, evaluated on 1000 samples.

Communication Path	Throughput (MB/s)	Latency (μ s)	Mechanisms Used	Payload size
Linux $\leftrightarrow$ FreeRTOS (ACK)	0.016 ± 0.0015	61.71 ± 6.125	OpenAMP (RPMMSG)	1 - 470 bytes
Linux $\leftrightarrow$ FPGA	$0.58 \pm 0.02 - 18.38 \pm 0.63$	$1.74 \pm 0.28 - 1.75 \pm 0.14$	GPIO	1 bit - 4 bytes
Linux $\leftrightarrow$ FPGA	0.21 ± 0.44	59.31 ± 10.13	AXI DMA	4 bytes
FreeRTOS $\leftrightarrow$ FPGA	$1.78 \pm 0.04 - 56.5 \pm 1.48$	$0.56 \pm 0.02 - 0.57 \pm 0.02$	GPIO	1 bit - 4 bytes
FreeRTOS $\leftrightarrow$ FPGA	2.15 ± 0.03	3.72 ± 0.06	AXI DMA	8 bytes

used for other measurements, the standard deviation is very high, reaching approximately twice the average value.

Additional insights can be obtained from the boxplot reported in Figure 12. While the interquartile range and the median remain constant across all configurations, the number and magnitude of outliers are very high in the first 200 samples and then progressively decrease. This behavior significantly affects the standard deviation and, to a lesser extent, the average latency, which becomes stable and significantly lower after the warm-up phase.

These results provides us with a first important result in the design of the architecture of the applications implemented in the SoC. Essentially, if a friendly user interface is needed, and it is implemented by using a full fledged operating system as Linux, the price to be payed is a significant latency if data have to be transferred to FreeRTOS. In addition, although the transfer is largely stable in terms of average latency, occasional outliers may occur, likely due to internal Linux mechanisms involved in the RPMMSG protocol. These mechanisms make this type of communication interface suitable for exchanging control commands between Linux and FreeRTOS, but not for high-volume data transfers. Therefore, it is preferable to transfer data directly to the FPGA. In addition, these latency values are particularly significant when compared to those of all the other test scenarios. In fact, every other scenario that includes data exchange with the FPGA, both from Linux and FreeRTOS, shows lower latency values even by orders of magnitude.

The Linux $\leftrightarrow$ FPGA experiments produced a significant performance improvement because the physical interconnect is much faster. The communication Path is based on a faster bus, thus the raw physical layer is capable of much higher speeds. Also, the GPIO is a parallel bus that allows the CPU to directly access the FPGA memory.

The FreeRTOS $\leftrightarrow$ FPGA experiments provided the fastest exchange. Data moves over dedicated bus on the chip. This is the fastest possible connection, with minimal physical delay. In fact, AXI is a high-performance on-chip bus protocol. Both ends have direct, low-latency access to the same memory block. The DMA controller is implemented in the PL, allowing data to move between the memory and the AXI stream without any CPU involvement.

As mentioned above, to evaluate the latency in the various test scenarios we used the minimum usable information units. However, to facilitate the comparison, in the cases related to

the use of the GPIO we also included experiments with longer packets, to compare the result with that obtained using the AXI4-Stream interface. This allows us to identify two other interesting results. First of all, if the data to be transmitted is highly fragmented into small blocks, the AXI4-Stream is not very efficient, since it requires the use of stuffing bits to obtain packets with the minimum length. Furthermore, even if from the latency point of view the various test scenarios show comparable results, significantly different results can be seen from the throughput point of view. In fact, it clearly appears that the use of the GPIO interface allows achieving significantly higher values.

However, this result is only partial, because it refers to the exchange of relatively short packets. To analyze a scenario in which large files need to be transferred, such as images, we performed experiments using longer packets. In this case, the emphasis on latency decreases, as the transfer time is dominant. However, the importance of throughput increases.

Fig. 13 shows the throughput achievable in the packet exchange between Linux and FreeRTOS for different payload size. The two graphs shown in the figure refer to two different situations. The first one, referred to as *Linux-FreeRTOS with ack*, consists in estimating the bandwidth of the Linux $\rightarrow$ FreeRTOS $\rightarrow$ Linux interaction in the Linux $\rightarrow$ FreeRTOS direction only, using 1-byte response packets that act as ACKs. The transmission of the acknowledgment packets is not included in the bandwidth calculation. The second approach, referred to as *Linux-FreeRTOS with full echo*, consists in estimating the bandwidth of a bidirectional exchange, in which the same packet, with a variable payload, is sent both in the Linux $\rightarrow$ FreeRTOS direction and in the reverse direction. Clearly, in this case the transported payload bytes are double compared to the case with the single ACK. The confidence intervals are shown in the figure, although their small value are difficult to appreciate at the scale of the graph.

As expected, the throughput increases with the packet length in both cases. However, it is noticeable that the trend is almost linear up to the maximum packet length. It was not possible to use longer packets due to the limits imposed by the RPMMSG protocol. The fact that no saturation effect is observed indicates that the communication capacity between the two cores is not fully exploited by RPMMSG, despite having been used in the kernel configuration in Linux. In fact, the absence of a plateau (saturation) in the throughput graph as packet length increases, interrupted only by the maximum

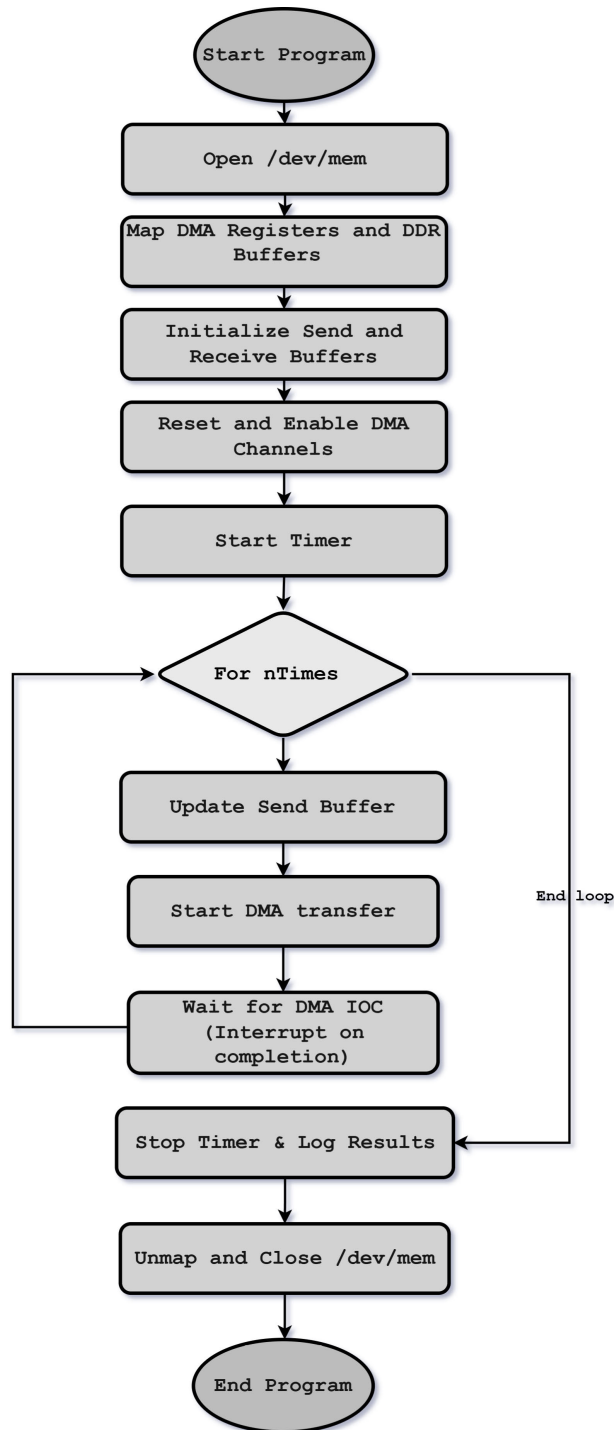


FIGURE 10. Flow diagram of the Linux-side C program used for AXI-Strem interfacing and data exchange with the FPGA.

limit imposed by RPMsg, indicates that the bottleneck is not in the transmission channel, but in the protocol. The system is therefore not operating at its maximum transmission capacity, which remains unexplored due to the protocol constraint.

Figure 14 shows the achievable throughput between PS and FPGA using GPIO. Again, in Figure 14 throughput exhibits

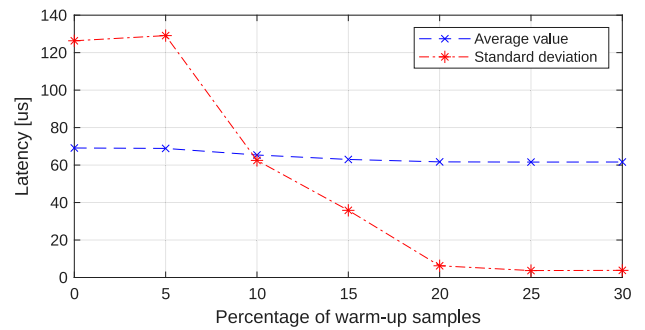


FIGURE 11. Measured latency for Linux → FreeRTOS → Linux (ACK): average and standard deviation as a function of the percentage of warm-up samples. The number of used samples is 1000.

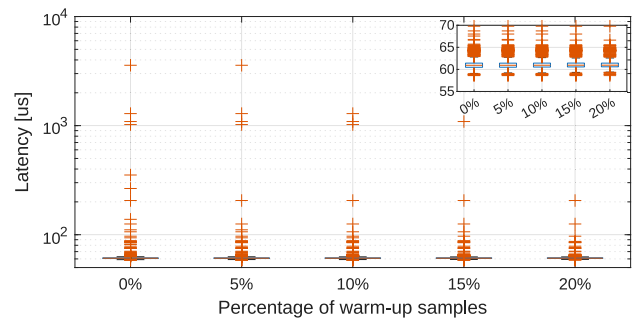


FIGURE 12. Boxplot of measured latency for Linux → FreeRTOS → Linux (ACK) as a function of the percentage of warm-up samples. The number of used samples is 1000. The inset plot focuses on inter-quartile ranges.

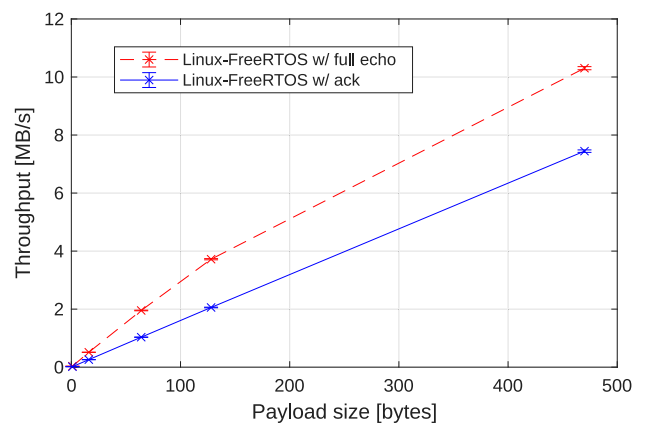


FIGURE 13. Measured bandwidth of the communication between Linux and FreeRTOS. ACK is not counted for throughput estimation. Number of iterations is equal to 1000.

an increasing trend, with response times that resulted almost independent of input and output size. The two graphs shown refer to PS applications deployed in Linux and FreeRTOS, respectively. The maximum values obtained by using the Linux application, referred to as *Linux-FPGA with GPIO*, doubles the average throughput values recorded for the Linux-FreeRTOS exchange (*Linux-FreeRTOS with full echo*). The maximum packet size shown in the figure is due to the maximum parallel bit transfer capacity of the GPIO. The

superior performance obtained using the FreeRTOS endpoint compared to the Linux endpoint is evident. In particular, the use of FreeRTOS in the PS allows the throughput that was observed in the Linux-based case to be tripled. This provides interesting implementation opportunities when it is essential to limit the effort and time of application development, since by using the GPIO interface the complexity of the code is appreciably lower than using the AXI Stream. The packet size, as in the case between Linux and the FPGA, does not depend on the RPMsg protocol, but is determined by the length allowed by the GPIO, which varies from 1 bit to 32 bits. These observations are a key aspect for making design decisions. Also in this case, the absence of a plateau in the throughput graph is due to the limited number of bits that the GPIO can handle. Due to the significance of the case study, including the PL, in Figure 15 we also show the latency values of the two solutions. It turns out that the latency values when the FreeRTOS endpoint is used are approximately one third compared to Linux endpoint, with a constant behavior as the number of bits exchanged in parallel through the GPIO interface varies. This confirms the advantage of using FreeRTOS for industrial applications where throughput and latency are key requirements.

The bandwidth measurement for Linux-FPGA communication using the AXI4-Stream protocol, shown in Figure 16, highlights appreciable improvements compared to using GPIOs for data transmission. GPIOs, although useful for simple applications that require the exchange of modest volumes of data, have limitations due to the overhead of synchronization operations and the reduced bandwidth. Instead, the plateau effect in Figure 16 shows that the adoption of the AXI4-Stream protocol, designed for continuous and high-speed data transfer, allows exploiting the actual capabilities of the FPGA. In this case the presence of a plateau in the throughput graph as the length of the packets increases indicates that the bottleneck is in the transmission channel, which is stressed by excess traffic, and that the system is therefore operating at its maximum transmission capacity which can therefore be considered determined. Also in this figure the intervals of confidence are included in the figure although they are difficult to appreciate at the scale of the graph.

The test was performed with packets of increasing size, from 8 Bytes to 8192 Bytes, to observe the impact of the payload size on the achievable throughput. The results indicate that the average bandwidth increases progressively with the increase of the payload size. The two graphs shown in the figure refer to the use of Linux and FreeRTOS, respectively, in the PS. Similar to what was observed in the use of the GPIO, when the exchange end-point in the PS is implemented with FreeRTOS the performance is significantly better than when using Linux. The difference in throughput is very significant, approximately 4 times. This difference provides a clear implementation guideline that was used in the implementation of the case study illustrated below. Also in this figure the intervals of confidence, although included, are difficult to appreciate in the figure at the scale of the graph.

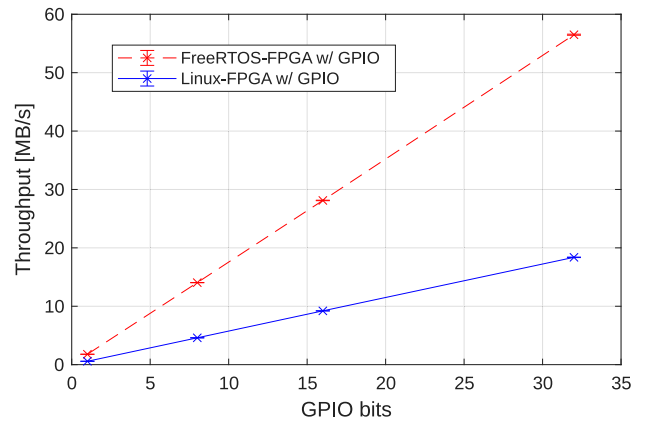


FIGURE 14. Measured bandwidth of the communication between PS and FPGA using GPIO. The FPGA implements a simple binary NOT function. ACK is not counted for throughput estimation. Number of iterations is equal to 1000.

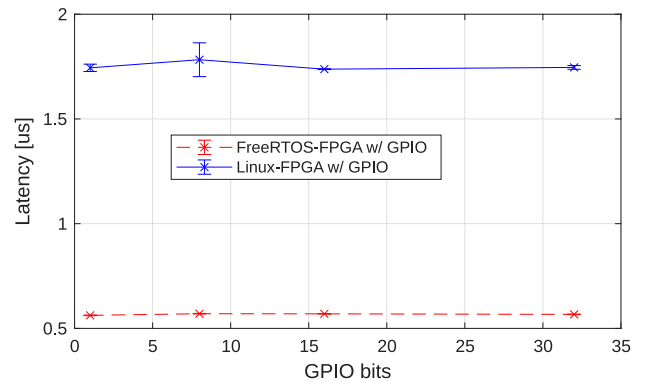


FIGURE 15. Measured latency of the communication between PS and FPGA using GPIO. The FPGA implements a simple binary NOT function. Number of iterations is equal to 1000.

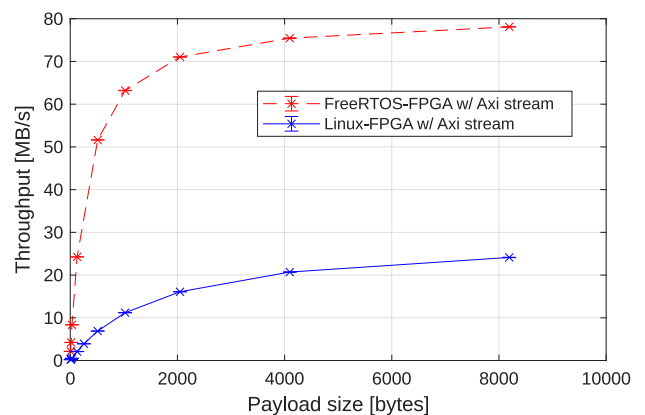


FIGURE 16. Measured bandwidth of the communication between Linux and FPGA using AXI-Stream. The FPGA implements a simple binary NOT function. ACK is not counted for throughput estimation. Number of iterations is equal to 1000.

Being this experiment the one providing the best throughput performance, also in this case we depict the latency for payload transfer, for both Linux and FreeRTOS endpoints, in Figure 17. As expected, the message transfer latency

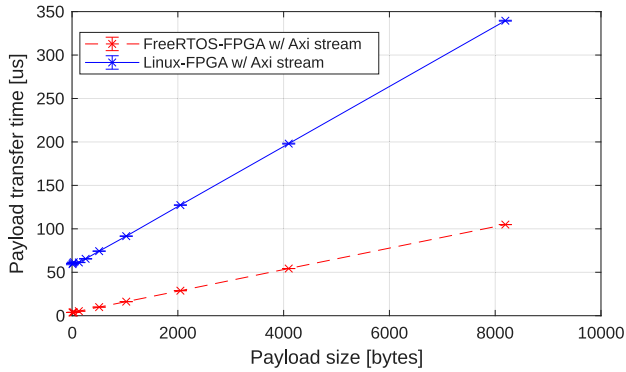


FIGURE 17. Measured latency of the communication between Linux and FPGA using AXI-Stream. The FPGA implements a simple binary NOT function. Number of iterations is equal to 1000.

increases almost linearly with the payload size, with a lower slope and intercept value when FreeRTOS is used.

VII. DISCUSSION AND SAMPLE APPLICATION

A. KEY INSIGHTS FOR EDGE COMPUTING

Using the results of the previous section we can draw conclusions regarding the service categories presented in Section II.

For Real Time applications, for which latency is the critical factor, it is necessary to limit data exchanges between the two cores of the SoC. If this must be done, either to ensure a friendly user interface or for the easy use of results, it should be limited to a single exchange. Otherwise, it is best to refer to the FPGA, using either the GPIO interface or the AXI4-Stream Interface.

- **Linux ↔ FreeRTOS:** Variable latency $> 60 \mu s$. Core isolation is not sufficient to further decrease latency. These values represent a significant decrease in the processing time available to implement applications with strict latency limits.
- **FreeRTOS ↔ FPGA:** $\leq 1 \mu s$ latency using GPIO, up to 32 bit payload size, and transfer time $\leq 4 \mu s$ for AXI4-Stream up to 32 byte payload size. Using the AXI4-Stream interface, latency increases nearly linearly until 8196 bytes (about $105 \mu s$), producing a significant throughput, around 80 MB/s. This solution is suitable for implementing safety systems, such as autonomous driving.
- **Linux ↔ FPGA:** In this case the considerations are completely similar to that of the exchange between Linux and FPGA, but with a bit increased latency and thus decreased throughput. This allows additional freedom to design the application logic and dictate the path the data should take internally to the SoC.

To sum up, our measurement campaign shows that it is possible to distinguish between protocol-limited communication throughput (e.g., RPMsg, which does not exhibit a clear throughput plateau) and link-limited throughput (e.g., AXI4-Stream, which exhibits a well-defined plateau). Therefore, the communication architecture plays a decisive role in

determining where the bottleneck lies, i.e., in the protocol or in the communication channel.

The arrival process of the information units to be exchanged requires additional considerations.

- **Continuous Delivery:** In this case, given the possibility of creating packets of the desired size, the *desired* trade-off on latency and throughput can be obtained using the AXI4-Stream interface.
- **Burst Traffic:** If it is necessary to transmit small packets, as happens in some IoT applications, given the measured efficiency values, it may be preferable to use the GPIO interface. If the length of the packets increases, even if of variable value, then it would be advisable to use the AXI4-Stream again.

Trade-offs specific in edge computing applications for real-time applications are summarized in Table 4.

For what concerns power consumption, the use AXI DMA for bulk data instead of CPU-driven transfers is effective to reduce active power. It is possible to put idle cores in low-power states (e.g., WFI - Wait For Interrupt) for this purpose. A possible pitfall is that shared memory polling keeps CPUs active, wasting power. An alternative solution is to Use interrupts (IRQ) to wake cores only when needed.

For obtaining hard Real-Time, the optimal solution is to leverage FreeRTOS ↔ FPGA via AXI4-Stream or AXI-Lite for message exchange. In fact, AXI registers allow sub-10 μs access and interrupts (IRQ) ensure immediate response. A possible pitfall is that Linux is non-real-time and even with the use of the well-known kernel patch *PREEMPT_RT* [16], latency can exceed 100 μs . It is therefore necessary to reserve a core for FreeRTOS.

For maximizing throughput, the suitable solution is to use AXI DMA (Linux ↔ FPGA), since it supports 64-bit bursts (up to 6 GB/s in Zynq-7000 bus), thus offloading data movement from CPUs.

In terms of flexibility, the usage of Linux is suitable for protocols as TCP/IP, MQTT, USB). FreeRTOS can be used for real-time control (motor PWM, sensor sampling). In this case, a possible pitfall is that FreeRTOS lacks dynamic features. In particular, no virtual memory is available and limited multi-threading is supported. Overloading it with complex tasks risks deadline misses.

B. SAMPLE INDUSTRIAL APPLICATION

We applied the results found in an edge computing sample application that can adequately demonstrate them. This application is of industrial nature, and consists of a manufacturing defect monitoring and alarm system. The schematic workflow of the application is shown in Figure 18.

This application uses a camera to capture video of the manufactured components passing on a conveyor belt, analyzes the individual frames of the components and searches for any manufacturing defects by scanning the individual frames with a trained deep convolutional neural network. When a defect is identified, an alarm is generated and the defect is framed in red

TABLE 4. Edge requirements for real-time applications vs. solutions.

Requirement	Optimal Solution	Pitfalls
Low power	Minimize Linux ↔ FPGA transfers	Avoid shared memory polling
Hard real-time	FreeRTOS/Linux ↔ FPGA	Bypass Linux ↔ FreeRTOS for critical paths
High bandwidth	AXI DMA (Linux ↔ FPGA)	RPMSG is too slow
Flexibility	Linux (TCP/IP) + FreeRTOS	Do not overload FreeRTOS

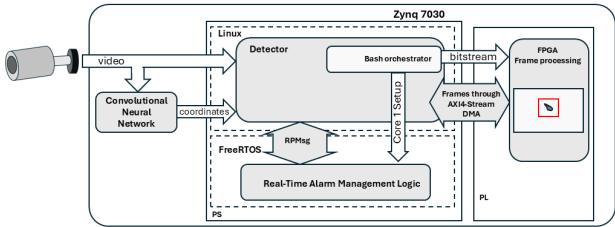


FIGURE 18. Schematic workflow of the sample application.

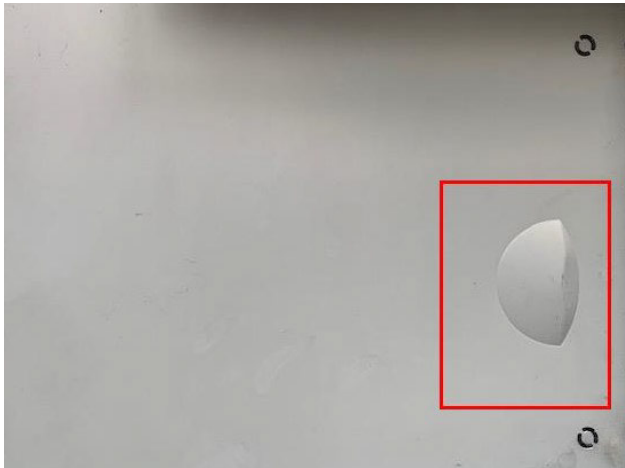


FIGURE 19. Example of a manufacturing defect framed in red by the FPGA. The Verilog code used for implementing this frame processing is reported in APPENDIX. The logs generated by Vivado with detailed view of the parameters illustrating the system behavior are included in the repository [1].

in the video. The repository [1] includes the detailed structure of the deep neural network used.

The application was designed by making use of the hardware design shown in Figure 9, modified by two custom IPs programming the FPGA. One of them makes use of the AXI4-Lite interface to send the box edge coordinates to the FPGA. The other one implements the Verilog code used to draw the box. When implemented in the FPGA, the function that draws the box around the defect requires a Look-Up Table (LUT) utilization of 4.39% and a Block RAM (BRAM) utilization of 1.13%.

Figure 19 shows an example of detected and highlighted manufacturing issue. In order to give the interested reader an example of the resulting code for the FPGA the Appendix reports the Verilog code used for drawing the red box.

From an implementation point of view, it emerged during the experiments that the most challenging component is the addition of the red box around the defect for each single frame.

If this operation is not done with a timing compatible with the frame rate, the resulting video is not smooth. We developed a Linux application using the open source WebRTC (Web Real-Time Communication) technology, which allows real-time communication between web browsers, without the need for plugins.

We considered the default frame rate, which is 30 fps. Therefore, in the case of identifying a defect, it is necessary to notify FreeRTOS to report the alarm and pass 30 frames every second to the FPGA to introduce the box in the defect position.

The central element of the application is the orchestrator. In fact, the flexibility of the Linux interface can be exploited through bash commands that, when necessary, activate the core slave, run the alarm management program in FreeRTOS, and configure the FPGA by loading the required bitstream file. Furthermore, the frames and the position of the defect are sent to the FPGA that takes care of realizing the most computationally expensive component of the service. The data exchange with the PL takes place through the AXI4-Stream interface.

Given the demanding and continuous image transfer, to demonstrate the feasibility of the application we only measured the number of frames that can be processed using the FPGA and the AXI4-Stream interface. Results are reported in Table 5 where different resolutions are considered for the transferred frames.

It is also important to note that in the absence of defects, which hopefully happens most of the time, core 1 and the FPGA are usable for other purposes. If we imagine a scenario of using a stack of SoCs for different application needs, this dynamicity allows the orchestrator to leverage statistical multiplexing to maximize the efficiency of using the available resources and minimize power consumption.

The results show that in the case of 640×480 image size, the latency specifications are largely satisfied. In the case of 1280×720 images we are at the limit of the specifications and there is no margin to use the FPGA for other purposes. For 1920×1080 images it is not possible to respect the necessary latency values. In addition, we have also done a comparison with a software-only edge node. The experiments have been carried out on a Thinkpad PC, equipped with an Intel® Core™ i9-11950H @2.60GHz processor (8 cores, 16 threads) and 128 GB of RAM, using the cv2.rectangle() function of OpenCV library to draw a rectangle for defect framing. Although we used well-sized hardware, obtained results were not satisfactory at all, also for small frames,

TABLE 5. Latency performance of defect framing by FPGA and CPU-only solution: average values with standard deviation (in ms), evaluated on 1000 and 200 samples for FPGA and CPU, respectively. Also achievable frame rates (in frame/s) are reported.

Image Size	FPGA Latency (ms)	FPGA rate (fps)	CPU Latency (ms)	CPU rate (fps)
640x480	10.21 ± 0.07	97.96	167.05 ± 39.29	5.99
1280x720	30.43 ± 0.23	32.86	501.15 ± 117.86	1.99
1920x1080	68.47 ± 0.52	14.6	1127.6 ± 262.19	0.89

as can be appreciated by values reported in Table 5. In fact, also for small images (640 × 480 pixels), the achievable frame rate is well below the target of 30 fps, which makes a software-only solution difficult to apply, even with more performing hardware.

This example allows it to critically analyze the obtainable performance. Indeed, the limitation for the relatively cheap SoC used in image processing has been highlighted. This also makes evident the trade-off between the cost of the device and the performance that can be achieved even if applying best practices in application implementation.

The results we have obtained will also be used for an ongoing activity aimed at creating a control system for industrial burn-in systems, which presents even more challenging aspects than those we faced.

VIII. CONCLUSION AND FUTURE WORK

In this paper we present implementation solutions for creating real time applications for edge computing. The basic assumption of the work is the use of a SoC belonging to the AMD Zynq 7000 family. This choice is due to the possibility of creating applications on a single chip, using two operating systems and an FPGA simultaneously. Specifically, we loaded the Linux and FreeRTOS operating systems, each into one core of the SoC. The former ensures we use user interface friendly libraries, the latter ensures real-time control. In this way, by also being able to use the hardware acceleration provided by the FPGA, we can create Real Time service architectures at reasonably low costs, with small footprint and reduced energy consumption.

To identify the best implementation strategies, we carried out a certain number of experimental tests to measure the latency and flow rate of data exchanges between the cores and between each core and the FPGA. In particular, for the data exchange between the two operating systems we used the RPMsg protocol, while we made use of the AXI GPIO and AXI4-Stream interfaces as extremal solutions for data exchange between the operating systems and the FPGA. This experimental analysis allowed us to identify selection criteria in the design of applications. We used these criteria to create an industrial application which showed the real functionality of our findings.

As future work, we plan to investigate the impact of Linux real-time tuning on communication latency with FreeRTOS and/or the PL, as latency is jointly determined by PS scheduling behavior, PL data paths, and the interaction between DMA engines and interrupt handling. Real-time kernel tuning there-

fore represents a complementary approach to further reduce latency variability and improve determinism.

For instance, the adoption of a PREEMPT_RT-enabled kernel is expected to directly affect the latency observed at the PS-PL boundary. On the Zynq-7030, AXI DMA completion interrupts generated in the PL are served by the ARM Cortex-A9 cores in the PS. Under a standard kernel, non-preemptible sections and hard IRQ handling may delay the execution of DMA completion handlers or user-space wake-ups, potentially increasing latency, as observed in some experiments. In contrast, PREEMPT_RT converts most interrupt handlers into preemptible kernel threads and shortens non-preemptible regions. As a result, it is expected to reduce worst-case latency and jitter without significantly affecting the mean value, which is dominated by AXI transfer and PL processing times.

The use of full tickless operation (nohz_full) on application-dedicated cores is particularly relevant for Zynq-based systems, where a limited number of CPU cores (only two on the Zynq-7030 SoC) has to simultaneously handle user-space real-time tasks, kernel threads, and interrupts generated by the PL. Periodic scheduler ticks can introduce regular timing noise that is comparable in magnitude to the communication latencies measured between PS and PL. Suppressing scheduler ticks on specific core(s) is therefore expected to reduce periodic jitter components in latency measurements, especially in steady-state streaming scenarios driven by the AXI DMA. CPU isolation techniques further extend this approach by preventing non-essential kernel activities from executing on cores responsible for real-time interaction with the PL. On the Zynq-7030 SoC, isolating one Cortex-A9 core for high priority tasks, such as DMA control, interrupt handling, and time-critical user-space threads, and assigning background tasks to the other core, is expected to improve both average communication latency and its variability.

Finally, by aligning IRQ affinity with CPU isolation, DMA-related interrupts can be handled on a core dedicated to real-time processing (as in the Zynq-7030 SoC) or intentionally separated from the main application thread, depending on the number of available cores. This approach is expected to further reduce latency variability due to interrupt handling.

APPENDIX A VERILOG CODE FOR SAMPLE APPLICATION

This appendix reports the Verilog code of a custom Vivado IP that we created to implement the indus-

trial sample application that we illustrated in the paper. In particular, this code can draw a red box in video frames in the position where an implementation defect is detected.

APPENDIX B

SYSTEM CONFIGURATION FOR SAMPLE APPLICATION

Our SoC (Zynq-7030) integrates the PS with an AXI DMA v7.1 instance configured in simple mode. The DMA is

```

`timescale 1ns / 1ps

module SquareBorder_Axi_Control(
    input wire          aclk,
    input wire          aresetn,
    input wire [63:0]   s_axis_tdata,
    input wire          s_axis_tvalid,
    output wire         s_axis_tready,
    input wire          s_axis_tlast,
    output reg [63:0]   m_axis_tdata,
    output reg          m_axis_tvalid,
    input wire          m_axis_tready,
    output reg          m_axis_tlast,

    // Runtime-configurable square border
    input wire [10:0]   square_x_start,
    input wire [10:0]   square_x_end,
    input wire [9:0]    square_y_start,
    input wire [9:0]    square_y_end
);

    localparam IMG_WIDTH      = 640;
    localparam IMG_HEIGHT    = 480;
    localparam PIXELS_PER_FRAME = IMG_WIDTH *
        IMG_HEIGHT;
    localparam BYTES_PER_PIXEL = 4;
    localparam TOTAL_BYTES    = PIXELS_PER_FRAME *
        BYTES_PER_PIXEL;
    localparam BLOCK_SIZE     = 2048;
    localparam [31:0] RED_RGBA = 32'hFF0000FF;

    reg [20:0] bytes_processed;
    reg [18:0] pixel_counter;
    reg [10:0] block_bytes;
    reg        frame_active;

    wire [10:0] x0 = pixel_counter % IMG_WIDTH;
    wire [10:0] x1 = (x0 == IMG_WIDTH - 1) ? 0 : x0
        + 1;
    wire [9:0] y = pixel_counter / IMG_WIDTH;

    assign s_axis_tready = m_axis_tready;

    function is_on_square_border;
        input [10:0] x;
        input [9:0] y;
        begin
            is_on_square_border = (
                ((x >= square_x_start) && (x <
                    square_x_end) &&
                ((y >= square_y_start && y <
                    square_y_start + 3) ||
                (y >= square_y_end - 3 && y <
                    square_y_end))) ||
                ((y >= square_y_start) && (y <
                    square_y_end) &&
                ((x >= square_x_start && x <
                    square_x_start + 3) ||
                (x >= square_x_end - 3 && x <
                    square_x_end)))
            );
        end
    endfunction

    always @(posedge aclk or negedge aresetn) begin
        if (!aresetn) begin
            bytes_processed <= 0;
            pixel_counter <= 0;
            block_bytes <= 0;
            frame_active <= 0;
            m_axis_tvalid <= 0;
            m_axis_tlast <= 0;
            m_axis_tdata <= 0;
        end else begin
            m_axis_tvalid <= 0;
            m_axis_tlast <= 0;

            if (s_axis_tvalid && s_axis_tready)
                begin
                    if (!frame_active) begin
                        frame_active <= 1;
                        bytes_processed <= 0;
                        pixel_counter <= 0;
                        block_bytes <= 0;
                    end

                    if (is_on_square_border(x0, y))
                        begin
                            m_axis_tdata[31:0] <= RED_RGBA;
                        end else begin
                            m_axis_tdata[31:0] <=
                                s_axis_tdata[31:0];
                        end

                    if (is_on_square_border(x1, y))
                        begin
                            m_axis_tdata[63:32] <= RED_RGBA;
                        end else begin
                            m_axis_tdata[63:32] <=
                                s_axis_tdata[63:32];
                        end

                    m_axis_tvalid <= 1;

                    if (m_axis_tready) begin
                        pixel_counter <= pixel_counter
                            + 2;
                        bytes_processed <=

```

LISTING 1. Verilog code for programming the FPGA in the sample application.

LISTING 1. (Continued.) Verilog code for programming the FPGA in the sample application.

```

        bytes_processed + 8;
        block_bytes <= block_bytes + 8;

    if (block_bytes + 8 >=
        BLOCK_SIZE) begin
        m_axis_tlast <= 1;
        block_bytes <= 0;
    end

    if (bytes_processed + 8 >=
        TOTAL_BYTES) begin
        m_axis_tlast <= 1;
        frame_active <= 0;
        bytes_processed <= 0;
        pixel_counter <= 0;
        block_bytes <= 0;
    end
end
end

if (s_axis_tlast && !frame_active)
    begin
        bytes_processed <= 0;
        pixel_counter <= 0;
        block_bytes <= 0;
    end
end
end

endmodule

```

LISTING 1. (Continued.) Verilog code for programming the FPGA in the sample application.

instantiated with memory-to-stream (MM2S) and stream-to-memory (S2MM) support enabled, scatter-gather functionality disabled, and a 64-bit AXI memory interface. This configuration is representative of many image-processing and data-streaming applications where data flows from DDR memory in the PS to the PL. In this context, explicit cache maintenance on the PS is mandatory. The Cortex-A9 data caches in Zynq-7000 devices are not hardware-coherent with AXI masters in the PL, including the AXI DMA. Thus, any data buffer written by the CPU and subsequently accessed by the DMA must be explicitly flushed from the data cache to main memory before starting the transfer. Otherwise, the DMA may read stale data from DDR, since modified cache lines may not yet have been written back. In the MM2S-only configuration, the correct software sequence therefore consists of (i) populating the transmit buffer in DDR, (ii) performing a cache flush over the corresponding address range, (iii) programming the DMA source address and transfer length, and finally (iv) starting the DMA engine. We disabled the Scatter-Gather feature in both DMA instances, namely `axi_dma_0` (MM2S) and `axi_dma_1` (S2MM), by setting the `C_INCLUDE_SG` parameter to 0. As a result, the DMA engines operate in Direct Register mode rather than using a descriptor ring. In this mode, the software initiates transfers by writing the physical address of the buffer directly to the

SRCADDR (for MM2S) or DESTADDR (for S2MM) registers, followed by the transfer length in bytes to the LENGTH register.

Synchronization between software and the DMA engine is implemented through polling. This approach avoids the complexity of configuring the generic interrupt controller and simplifies control flow, at the cost of increased CPU resource consumption. In practice, the software continuously reads the DMA Status Register (DMASR) and check the `IOC_Irq` (Interrupt on Complete) bit. When this bit is asserted, the transfer is complete. The software then can clear the bit by writing a '1' back to it before initiating the next operation.

REFERENCES

- [1] *Public Repository With Reproducible Code and Vivado/Vitis Scripts*. Accessed: Nov. 13, 2025. [Online]. Available: https://conan.diei.unipg.it/zynq/zynq_archive.zip
- [2] S. Alonso, J. Lazaro, J. Jimenez, L. Muguira, and U. Bidarte, "Evaluating the OpenAMP framework in real-time embedded SoC platforms," in *Proc. XXXVI Conf. Design Circuits Integr. Syst. (DCIS)*, Nov. 2021, pp. 1–6.
- [3] Altera. (2015). *Arria V SoC Development Board Reference Manual*. [Online]. Available: https://cdrdv2-public.intel.com/653820/rm_av_soc_dev_board.pdf
- [4] AMD. *Package Installation Problem—Discuss.PYNQ.IO*. Accessed: May 8, 2025. [Online]. Available: <https://discuss.pynq.io/t/package-installation-problem/5312>
- [5] AMD. *PYNQ—PYNQ.IO*. Accessed: May 8, 2025. [Online]. Available: <https://www.pynq.io/>
- [6] AMD. (2024). *AMD Technical Information Portal—Docs.Amd.Com*. Accessed: May 23, 2025. [Online]. Available: <https://docs.amd.com/r/en-U.S./ug1165-zynq-embedded-design-tutorial/Vitis-Software-Platform-and-Vivado-Design-Suite>
- [7] AMD. (2024). *AMD Technical Information Portal—Docs.Amd.Com*. Accessed: May 17, 2025. [Online]. Available: <https://docs.amd.com/r/en-U.S./ug1186-zynq-openamp-gsg/OpenAMP-Demos>
- [8] AMD. (2025). *AMD Zynq 7000 SoCs: SoCs With Hardware and Software Programmability*. Accessed: May 22, 2025. [Online]. Available: <https://www.amd.com/en/products/adaptive-socs-and-fpgas/soc/zynq-7000.html>
- [9] AMD. (2025). *Getting Started With Versal Memory Interfaces*. [Online]. Available: https://adaptivesupport.amd.com/s/article/1192335?language=en_US
- [10] AMD Xilinx, *Zynq-7000 SoC Technical Reference Manual (UG585)*, AMD, v1.14.1 edition, Sunnyvale, CA, USA, Dec. 2023.
- [11] Tilottoma Barua. (2020). *Data Transfer Performance Analysis From Programmable Logic to Processing System of Zynq 7000—Hammer.Purdue.Edu*. Accessed: May 2, 2025. [Online]. Available: https://hammer.purdue.edu/articles/thesis/DATA_TRANSFER_PERFORMANCE_ANALYSIS_FROM_PROGRAMMABLE_LOGIC_TO_PROCESSING_SYSTEM_OF_ZYNQ_7000/12743315?file=24115658
- [12] J. Boudjadar, S. U. Islam, and R. Buyya, "Dynamic FPGA reconfiguration for scalable embedded artificial intelligence (AI): A co-design methodology for convolutional neural networks (CNN) acceleration," *Future Gener. Comput. Syst.*, vol. 169, Aug. 2025, Art. no. 107777.
- [13] T. P. Dinh, C. Pham-Quoc, T. N. Thinh, B. K. D. Nguyen, and P. C. Kha, "A flexible and efficient FPGA-based random forest architecture for IoT applications," *Internet Things*, vol. 22, Jul. 2023, Art. no. 100813.
- [14] P. Dong, A. Burns, Z. Jiang, and X. Liao, "Tzdk: A new trustzone-based dual-criticality system with balanced performance," in *Proc. IEEE 24th Int. Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA)*, Aug. 2018, pp. 59–64.
- [15] P. Dong, Z. Jiang, A. Burns, Y. Ding, and J. Ma, "Build real-time communication for hybrid dual-OS system," *J. Syst. Archit.*, vol. 107, Aug. 2020, Art. no. 101774.
- [16] The Linux Foundation. (2025). *Realtime: Start [Wiki]—Wiki.Linuxfoundation.Org*. Accessed: May 23, 2025. [Online]. Available: <https://wiki.linuxfoundation.org/realtime/start>

- [17] L. F. L. França, F. Benevenuti, L. R. Gobatto, R. P. Bastos, J. R. F. Azambuja, and F. L. Kastensmidt, "Open source port of NASA's core flight system to FreeRTOS in arm cortex-M-based microprocessor," in *Proc. IEEE 16th Latin Amer. Symp. Circuits Syst. (LASCAS)*, Feb. 2025, pp. 1–5.
- [18] A. M. García, C. O. Quero, J. Rangel-Magdaleno, J. Martinez-Carranza, and D. D. Romero, "Edge computing SoC implementation of compressive sensing algorithm for single-pixel cameras," in *Proc. 18th Int. Conf. Electr. Eng., Comput. Sci. Autom. Control (CCE)*, Nov. 2021, pp. 1–5.
- [19] M. Hermassi, S. Krim, Y. Krim, M. A. Hajjaji, A. Mtibaa, and M. F. Mimouni, "Xilinx-FPGA for real-time implementation of vector control strategies for a grid-connected variable-speed wind energy conversion system," in *Proc. 5th Int. Conf. Adv. Syst. Emergent Technol. (IC_ASET)*, Mar. 2022, pp. 49–54.
- [20] H. G. M. Hernandez, F. Fricke, M. Al Kadi, M. Reichenbach, and M. Hübner, "Edge GPU based on an FPGA overlay architecture using PYNQ," in *Proc. 35th SBC/SBMicro/IEEE/ACM Symp. Integr. Circuits Syst. Design (SBCCI)*, Aug. 2022, pp. 1–6.
- [21] A. Hernandez, R. Nieto, D. Fuentes, and J. Urena, "Design of a SoC architecture for the edge computing of NILM techniques," in *Proc. XXXV Conf. Design Circuits Integr. Syst. (DCIS)*, 2020, pp. 1–6.
- [22] O. Hryshchuk and S. Zagorodnyuk, "Managing energy consumption in FPGA-based edge computing systems with soft-core CPUs," *J. Edge Comput.*, vol. 4, no. 1, pp. 57–72, May 2025.
- [23] T. Inage et al., "M-KUBOS/PYNQ cluster for multi-access edge computing," in *Proc. 9th Int. Symp. Comput. Netw. (CANDAR)*, Nov. 2021, pp. 95–101.
- [24] Intel Corporation. (2023). *Cyclone V Featured Documentation—Quick Links Guide*. [Online]. Available: <https://www.intel.com/content/www/us/en/docs/programmable/783856/current/featured-documentation-quick-links.html>
- [25] Z. Li, Y. Hao, H. Gao, and J. Zhou, "Towards intelligent edge computing: A resource- and reliability-aware hybrid scheduling method on multi-FPGA systems," *Electronics*, vol. 14, no. 1, p. 82, Dec. 2024.
- [26] OpenAMP. *Inter Process Communications (IPC) Through RPMsg Protocol; OpenAMP Documentation*. Accessed: May 3, 2025. [Online]. Available: https://openamp.readthedocs.io/en/latest/protocol_details/rpmmsg.html
- [27] OpenAMP. *OpenAMP—Github.Com*. Accessed: May 23, 2025. [Online]. Available: <https://github.com/openamp>
- [28] OpenAMP. *RPMsg Protocol Layers; OpenAMP Documentation—Openamp.Readthedocs.IO*. Accessed: May 23, 2025. [Online]. Available: https://openamp.readthedocs.io/en/latest/protocol_details/rpmmsg_protocol.html#transport-layer-rpmmsg
- [29] A. Pandya, R. Agrawal, V. Panchal, and R. Parekh, "ASIC SoC design for edge computing agriculture application," in *Proc. 11th Int. Conf. Signal Process. Integr. Netw. (SPIN)*, Mar. 2024, pp. 105–110.
- [30] G. Perkins et al., "SoK: Trusted execution in SoC-FPGAs," 2025, *arXiv:2503.16612*.
- [31] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral, "LTZVisor: Trust-Zone is the key," in *Proc. 29th Euromicro Conf. Real-Time Syst. (ECRTS)*, Dagstuhl, Germany, 2017, pp. 4:1–4:22.
- [32] A. A. Powell. (2014). *Performance of the Xilinx Zynq System-on-Chip Interconnect With Asymmetric Multiprocessing—Scholarshare.Temple.Edu*. Accessed: May 2, 2025. [Online]. Available: <https://scholarshare.temple.edu/items/ef015518-1a73-4af0-9166-a7f716004e7b>
- [33] D. Sangorrín, S. Honda, and H. Takada, "Reliable device sharing mechanisms for dual-OS embedded trusted computing," in *Proc. Int. Conf. Trust Trustworthy Comput.*, vol. 7344, 2012, pp. 74–91.
- [34] P. Schulz and G. Sleatitichi, "FPGA-based accelerator method for edge computing," Hamburg Univ. Appl. Sci., Hamburg, Tech. Rep., 2024, doi: [10.20944/preprints202405.0657.v1](https://doi.org/10.20944/preprints202405.0657.v1). [Online]. Available: <https://reposit.haw-hamburg.de/bitstream/20.500.12738/16697/1/preprints202405.0657.v1-1.pdf>
- [35] V. Sharma, "Managing multi-cloud deployments on kubernetes with istio, prometheus and grafana," in *Proc. 8th Int. Conf. Adv. Comput. Commun. Syst. (ICACCS)*, vol. 1, Mar. 2022, pp. 525–529.
- [36] Y. Shu, Y. Matsubara, Y. Li, and H. Tekada, "Towards a linux-based unikernel for resource-constrained embedded systems," in *Proc. Preliminary 19th Annu. Workshop Operating Syst. Platforms Embedded Real-Time Appl. (OSPRT)*, 2025, pp. 23–27.
- [37] P. Soma and R. K. Jatoth, "Hardware implementation issues on image processing algorithms," in *Proc. 4th Int. Conf. Comput. Commun. Autom. (ICCCA)*, Dec. 2018, pp. 1–6.
- [38] Y. Sun, E. Li, G. Yang, Z. Liang, and R. Guo, "Design of a dual-core processor based controller with RTOS-GPOS dual operating system," in *Proc. IEEE Int. Conf. Mechatronics Autom. (ICMA)*, Aug. 2019, pp. 1859–1864.
- [39] Xilinx. (2022). *Zynq Ultrascale+ MPSoC Data Sheet: Overview*. https://www.amd.com/content/dam/xilinx/support/documents/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf
- [40] M. Xu, L. Zhang, and S. Wang, "Position paper: Renovating edge servers with ARM SoCs," in *Proc. IEEE/ACM 7th Symp. Edge Comput. (SEC)*, Dec. 2022, pp. 216–223.
- [41] C.-F. Yang and Y. Shinjo, "Compounded real-time operating systems for rich real-time applications," *IEEE Access*, vol. 13, pp. 26079–26104, 2025.
- [42] Yocto. (2025). *The Yocto Project—Yoctoproject.Org*. Accessed: May 23, 2025. [Online]. Available: <https://www.yoctoproject.org/>



SARA CUCCAGNA received the degree in electronics and computer science engineering from the University of Perugia, in 2025. She is currently a Contract Researcher with the Department of Engineering, University of Perugia, working on design and implementation of embedded systems for stress test application based on systems on chip. Her research interests include the development of applications in a multi-core embedded systems.



MAURO FEMMINELLA (Member, IEEE) received the master's and Ph.D. degrees in electronic engineering from the University of Perugia, Italy, in 1999 and 2003, respectively. Since July 2022, he has been an Associate Professor with the Department of Engineering, University of Perugia. He is also a Representative with the Consortium CNIT, University of Perugia. He has co-authored more than 120 papers in international journals and refereed international conferences. His current research interests include molecular communications, big data systems, and network management solutions for 5G/6G networks.



GIANLUCA GIUSEPPONI received the degree in electronic engineering from the University of Perugia, Italy, in 1997. Since 1997, he has been with ELES, where he holds the role of a System Architect and a Software Developer. He designs test and reliability systems for semiconductor devices. He is a Senior Software Programmer specializing in embedded systems programming. He also specialized in multithreading and multiprocessor programming.



BARBARA GUASTICCHI received the master's degree in electronic engineering from the University of Perugia, Italy, in 1999. Since 2000, she has been with the Semiconductor Industry. After ten years spent with the ELES Application Group as a Senior Application Engineer, focusing on memory product testing, she currently serves as the Program Manager with ELES. She is responsible for coordinating the projects concerning the new reliability test system development.



GIANLUCA REALÌ (Member, IEEE) received the Ph.D. degree in telecommunications from the University of Perugia, Italy, in 1997. From 1997 to 2004, he was a Researcher with the Department of Electronic and Information Engineering, University of Perugia. In 1999, he visited the Computer Science Department, UCLA. Since January 2005, he has been an Associate Professor with the Department of Engineering, University of Perugia. His research interests include resource allocation over packet networks, wireless networking, network management, multimedia services, big data management, and nanoscale communications.

...



LUCA MORICONI (Member, IEEE) received the degree in electronic engineering from the University of Perugia, Italy. He is currently the Research and Development Manager and the Solution Roadmap Manager at ELES, where he is responsible for defining and implementing the technical roadmap of ELES solutions. He has more than 25 years of experience in the semiconductor reliability and test industry. Throughout his career, he has contributed to the development of ELES test

systems and solutions across various IC families, supporting customers in improving reliability, quality, and productivity.