Real-Time Systems
https://doi.org/10.1007/s11241-023-09402-4

Supporting AI-powered real-time cyber-physical systems on heterogeneous platforms via hypervisor technology

Edoardo Cittadini[1] · Mauro Marinoni[1] · Alessandro Biondi[1] · Giorgiomaria Cicero[1] · Giorgio Buttazzo[1]

Accepted: 16 June 2023

Abstract

The heavy use of machine learning algorithms in safety-critical systems poses serious questions related to safety, security, and predictability issues, requiring novel architectural approaches to guarantee such properties. This paper presents an architecture solution that leverages heterogeneous platforms and virtualization technologies to support AI-powered applications consisting of modules with mixed criticalities and safety requirements. The hypervisor exploits the security features of the Xilinx ZCU104 MPSoCs to create two isolated execution environments: a high performance domain running deep learning algorithms under the Linux operating system and a safety-critical domain running control and monitoring functions under the freeRTOS real-time operating system. The proposed approach is validated by a use case consisting of an unmanned aerial vehicle capable of tracking moving targets using a deep neural network accelerated on the FGPA available on the platform.

Keywords Hypervisor-based architecture · FPGA acceleration · AI acceleration · Real-time application · Multi-domain application

✉ Edoardo Cittadini
edoardo.cittadini@santannapisa.it

Mauro Marinoni
mauro.marinoni@santannapisa.it

Alessandro Biondi
alessandro.biondi@santannapisa.it

Giorgiomaria Cicero
giorgiomaria.cicero@santannapisa.it

Giorgio Buttazzo
giorgio.buttazzo@santannapisa.it

[1] Department of Excellence in Robotics & AI, Scuola Superiore Sant'Anna, Pisa, Italy

Published online: 17 July 2023

1 Introduction

Modern cyber-physical systems (CPS), e.g., cars, aircrafts, advanced robots, and drones, are characterized by an increasing complexity that calls for new technologies and architectural solutions to guarantee predictability, safety, and security requirements. In addition, the increased level of autonomy specified for such systems requires the adoption of artificial intelligence (AI) and, more specifically, machine learning algorithms, which in turn imply heavy use of hardware acceleration to satisfy the stringent real-time constraints imposed by the applications.

Unfortunately, however, today's AI algorithms are not ready to be integrated in mission-critical CPS, since their results cannot be always trusted and well-accepted engineering methodologies to mitigate the problem are still missing. A promising solution consists in coupling AI models with a set of classical algorithms that can take over the control of the system whenever the outputs produced by AI are not deemed safe, with the aim of bringing the system into fail-safe or fail-operational conditions.

In such complex systems, at least two groups of software components can be distinguished, being characterized by different sets of requirements and criticality levels:

- Software components that require support from a rich execution environment (e.g., based on the Linux operating system), like AI algorithms, acquisition and processing stacks for complex sensors (as cameras and LiDARs), and high-speed network communication services.
- Software components that require a high-integrity execution environment (e.g., powered by a real-time operating system), like low-level control functions, safety-critical monitoring activities, and procedures to ensure fail-safe/fail-operational behavior.

The components belonging to the first group can be deemed *not critical* for safety and security, provided that they are properly isolated from the *critical* ones belonging to the second group. In this context, strong isolation is required to ensure that non-critical components cannot affect the execution of critical ones, including the guarantee that cyber-attacks and faults cannot propagate from the former to the latter.

Isolation could be achieved by executing these software components on different hardware platforms, e.g., reserving an independent platform to host the execution of critical software only. However, in several cases, as for battery-operated flying drones, such features have to be provided under stringent resource constraints, imposing additional limitations in terms of space, weight, power, and cost (SWaP-C). For this reason, a more appropriate solution is to host the execution of software components with mixed and independent safety and security levels on the same hardware platform. These systems are also referred to as *mixed-criticality software systems* and can leverage hypervisor technology to enforce isolation as well as enable the execution of multiple operating systems on the same hardware.

Mixed-criticality systems powered by hypervisor technology have been investigated since many years from different perspectives, especially in domains such as avionics (Gaska et al. 2011), aerospace (Crespo et al. 2009), and control (Crespo et al. 2018). Farrukh and West (2022) proposed a hypervisor-based architecture for combining a Linux domain with a real-time critical domain on the same platform for a drone application, but no machine learning and hardware acceleration were exploited. Scordino et al. (2020) presented a modular hypervisor-based platform for industrial automation for integrating both real-time control code and software design tools, but no AI algorithms and FPGA acceleration were employed.

Similarly, the challenges of achieving real-time performance in AI-powered cyber-physical systems have been discussed and reviewed by several authors (Musliner et al. 1995; Radanliev et al. 2020; Seng et al. 2021). For instance, Wang and Luo (2022) presented a review on the optimal design of neural networks on FPGA platforms. Ji et al. (2021) presented the implementation of a deep neural network for real-time object detection and tracking on an embedded system based on an FPGA Zynq platform. Sciangula et al. (2022) proposed an efficient method for accelerating deep neural networks for autonomous driving applications on an FPGA-based SoC. All these works, however, did not leverage hypervisor technology to integrate mixed criticality components.

A conceptual hypervisor-based architecture for supporting the execution of complex functionalities that are typical of AI-enabled CPS was proposed by Biondi et al. (2020); however, to the best of our records, a practical solution that integrates the acceleration of deep neural networks with real-time control on a single platform using hypervisor technology is still missing.

1.1 Contribution

This work presents a concrete software architecture for supporting AI-enabled CPS with mixed-criticality components. The proposed architecture targets heterogeneous computing platforms that couple asymmetric multicores with programmable logic (FPGA). It leverages hypervisor technology with strong isolation, hardware acceleration of AI algorithms implemented in programmable logic, and monitoring strategies to take over the control of the system whenever non-critical software components fail, are attacked, or produce results that are deemed unsafe. The architecture is then specialized for the case of autonomous flying drones, showing how it can be used to build a safe and secure tracking application.

1.2 Paper organization

The rest of the paper is organized as follows: Sect. 2 discusses some relevant related work; Sect. 3 presents the general architectural approach; Sect. 4 describes how the proposed architecture has been instantiated to a specific use case consisting of a visual tracking application performed by a drone; Sect. 5 reports some experimental results; and, finally, Sect. 6 concludes the paper and presents some future work.

2 State of the art

To the best of our records, there is no established approach in the literature for developing AI-powered cyber-physical systems; rather, several architectural solutions have been proposed by researchers in different contexts.

A classical approach to handle functions with different criticality requirements consists in executing them on separate computing platforms, typically managed by different operating systems that communicate through an external link, such as a serial line, a CAN network, or an Ethernet bus. Examples of CPS that adopt this approach are the Intel Ready-to-Fly (RTF) Drone (Intel Corporation), the Cube Autopilot (CubePilot), and several solutions that leverage the Robot Operating System (ROS) (Gutiérrez et al. 2018) for drone autopilots (Liu et al. 2017) and autonomous cars (Reke et al. 2020). This approach, although practical, is not ideal for battery-operated systems with stringent requirements in terms of size, weight, power, and costs (SWaP-C), because it tends to introduce duplicates of computational resources and hardware components. Another problem with this approach is that the communication between the two platforms requires additional devices, which are generally slower than on-chip communication and also prone to faults and noise.

An alternative approach adopted, for instance, by the PX4 autopilot stack (Meier et al. 2015) and the F1tenth autonomous car (O'Kelly et al. 2019), leverages the Linux operating system to control devices using a standard POSIX API and executing real-time functions as high-priority threads. This solution has the advantage of using a single hardware platform, at the risk of exposing the critical subsystem to several threats related to safety, security, and timing predictability due to the lack of strong isolation among different processes in Linux.

A partitioning operating system (OS) (e.g., LynxOS), despite its interesting capabilities and suitability for supporting domain-specific standards such as ARINC-653 and AUTOSAR (see Leiner et al. 2007), does not properly solve these issues, especially because it typically supports software systems with limited heterogeneity.

The increasing demand for advanced features in embedded systems requiring the need for a rich OS, such as Linux, combined with the need of guaranteeing safety, security mechanisms, and real-time constraints, led to the adoption of *hypervisor* technology as the most effective solution for supporting mixed-criticality software.

In AI-powered CPS, this solution allows isolating AI algorithms, which commonly call for the need of hardware acceleration managed by the rich OS, from the critical functions, which can instead be handled by a real-time operating system. The following subsections revise the literature related to hypervisor-based approaches and hardware acceleration for deep neural networks.

2.1 Hypervisor-based solutions

Type-1 hypervisors are considered a preferable choice for the proposed architecture thanks to their small code base and better hardware control, which can guarantee high levels of security, safety, and time predictability. Furthermore, Type-1 hypervisors are more capable of guaranteeing predictable virtualization-related latency

(e.g., in response to interrupts), inter-domain communication, and scheduling of virtual machines.

Several Type-1 hypervisors are available and some relevant research work has been done to integrate them for executing specific CPS applications, which are reviewed next. Klein et al. (2018) proposed a solution based on seL4 to separate trusted and untrusted software in a UAV platform, focusing on the issue of guaranteeing security and isolation in the system. Almeida and Prochazka (2009) presented a solution based on PikeOS (SYSGO) to provide safe and secure partitioning for integrated modular avionics (IMA) in spacecraft applications. Craveiro et al. (2009) proposed to use the partitioning features of ARINC 653 in Space Real-Time Operating System (AIR) for providing isolation in the development of aerospace systems, but only for IA-32 and Sparc architectures. Pérez and Gutiérrez (2016); Pérez et al. (2016) implemented a real-time publish-subscribe communication mechanism in the Xstratum (Crespo et al. 2009) hypervisor integrating ARINC-653 with the Data Distribution Service (DDS). Biondi et al. (2021) proposed a hypervisor-based architecture for safety-critical embedded systems providing both time/memory isolation, security, real-time communication channels, as well as I/O virtualization to allow different virtual machines to share the peripheral devices. Farrukh and West (2022) proposed a hypervisor-based solution characterized by low overheads in accessing resources. Their approach requires strict time guarantees for both domains, forcing the execution of Linux on a single core with `SCHED_DEADLINE` (Lelli et al. 2016), which is a viable solution in terms of real-time constraints, but can introduce several limitations in the implementation of complex AI-based solutions.

2.2 Hardware acceleration

A peculiar feature of AI-powered cyber-physical systems is their massive computational workload for executing AI algorithms such as deep neural networks. The most demanding functions of these algorithms need to be accelerated on specific hardware, such as general-purpose graphics processing units (GPUs) or field-programmable gate arrays (FPGA), to satisfy real-time requirements.

Modern GPU-based heterogeneous platforms benefit from powerful and mature software support to accelerate AI algorithms, which allows the user to significantly contain the effort for achieving efficient implementations of tasks like object detection, image segmentation, and tracking. The acceleration frameworks for GPU-based platforms also allow a developer to seamlessly use, with no or just a few modifications, the AI models available in frameworks such as Tensorflow, PyTorch, and Caffe, even with the native parameters with floating-point precision.

On the other hand, when compared to FPGA-based platforms, GPU-based platforms are very demanding in terms of power consumption and struggle in providing a high degree of time predictability for hardware acceleration. Their power consumption can be one order of magnitude larger than the one required by FPGA-based platforms (Sciangula et al. 2022; Qasaimeh et al. 2019). Furthermore, as observed by Cavicchioli et al. (2017), GPU acceleration introduces highly variable

delays that cannot easily be bounded a priori, also due to the contention occurring on shared memory in the case of memory-intensive GPU tasks. As such, GPU-based are not the ideal solution for battery-operated CPS such as drones.

Besides being characterized by less energy consumption, FPGAs provide a very predictable execution behavior with respect to GPUs for hardware acceleration.

Two main approaches are used to accelerate deep neural networks by means of FPGA technology:

1. The synthesis of a network-specific accelerator provides the best performance but suffers from poor flexibility and scalability, especially for large networks. To name one of the most relevant issues, this approach ends up in deploying replicated logic that implements the same operation (e.g., convolutions) on different data. Some tools provide IPs (e.g., HLS4ML Fahim et al. 2021; AMD Xilinx: FINN) as standalone Verilog/VHDL entities, which can be later integrated into more complex designs. Another relevant limitation of this approach is that the generated IP must be entirely rebuilt every time there is any change in the network.
2. A more flexible solution is to accelerate neural networks by means of a dedicated softcore. For instance, Xilinx provides a Deep Learning Processor Unit (AMD Xilinx: DPU) as a library component in the Vitis-AI environment (AMD Xilinx: Vitis AI). Besides the evident benefits in terms of flexibility provided by a network-agnostic accelerator such as the DPU, an advantage of this approach is that a single DPU can concurrently accelerate multiple networks, while in the other approach, the number of networks that can be accelerated is mainly limited by the amount of FPGA resources (such as LUTs).

The main disadvantage of FPGAs is that they require a larger programming effort than GPUs, especially when developing a network-specific accelerator. Another restriction is the limited amount of FPGA resources that is available in several embedded platforms, which calls for the usage of dynamic partial reconfiguration (Biondi et al. 2016; Seyoum et al. 2021) of the FPGA at the cost of additional delays when serving acceleration requests. Furthermore, as observed by some authors (Vaishnav et al. 2018; Happe et al. 2015; Rupnow et al. 2009), even without dynamic reconfiguration, sharing an FPGA among tasks managed by a preemptive scheduling policy is not trivial, due to the significant amount of time required to save the state of the device.

Fortunately, especially in the case of softcore accelerators such as the DPU, compilation and optimization frameworks are available to drastically simplify the deployment of accelerated neural networks. These frameworks employ pruning and quantization (Zhou et al. 2017) of the network parameters to achieve an efficient execution on the FPGA. The accuracy drop of these optimization processes was found not to be significant in several application scenarios (Gholami et al. 2022; Liang et al. 2021). The optimization algorithms dealing with the conversion from floating point to integer values are indeed now efficient enough to guarantee consistency in the transformation of the models from one platform

to another (GPU to FPGA) (Ding et al. 2019). This makes FPGA-based MPSoCs the ideal reference platform for resource-constrained embedded systems.

2.3 Limitations of previous work

The works reviewed in Sect. 2.1 share two main limitations: they are all tied to a specific application and do not address the support of AI algorithms with hardware acceleration. Furthermore, none of them investigated in details the usage of hypervisor technology to implement fail-safe/fail-operational control algorithms. Furthermore, despite the benefits of FPGA acceleration highlighted in Sect. 2.2, none of the works discussed in Sect. 2.1 considered this relevant technology.

This work advances the state of the art by presenting a general architecture for autonomous CPS that leverages FPGA-based embedded platforms. The performance and capabilities of the presented architecture are then evaluated for a visual tracking application implemented by an autonomous drone, putting particular emphasis on the role of the architecture in the development of fail-safe/fail-operational control algorithms.

3 System architecture

The proposed architecture is composed of two isolated execution domains (i.e., virtual machines): a non-critical, high-performance domain running a rich general-purpose operating system (GPOS) and a critical domain running a real-time operating system (RTOS). The two domains can communicate, depending on the task they have to perform, using a set of services provided by the hypervisor. This solution has the advantage of supporting all those embedded real-time applications that require the use of a GPOS for implementing complex high-level functions, including AI algorithms, but also needs to guarantee safety, security, and real-time performance requirements for a subset of safety-critical functions.

In the proposed architecture, the GPOS serves the purpose of handling high-level tasks (e.g., rich communication stacks, processing of high-performance sensors such as RGB/depth cameras and 3D LiDARs, application frameworks, inference of neural networks) that typically lack support in real-time operating systems or present a complexity that could jeopardize the stringent requirements of critical activities. Hypervisor-assisted health monitors are provided to detect and stimulate reactions to software faults or cyber-attacks.

The architecture also exploits hardware acceleration, implemented on FPGA by means of a machine-learning-specific accelerator, to speed up computations and offload CPUs, thus allowing the computing system to perform other tasks in parallel, increasing throughput while reducing latencies. On the other hand, the RTOS domain takes care of low-level control, actuation commands, validation, and safety monitoring, directly communicating with the hardware. Note that these tasks may be possibly related to safety-critical functions.

Finally, the FPGA can also be used to deploy ad-hoc devices, with the result of increasing execution predictability, reducing overhead, and helping satisfy SWaP-C constraints by limiting the use of external devices.

The hypervisor allows enabling the co-existence of one or more GPOS and RTOS domains on the same platform and, most importantly, is required to ensure strong isolation between the two domains to guarantee high degrees of safety, security, and time-predictability for the critical domain. If some of the physical resources are shared across different domains, the hypervisor uses a scheduling policy based on time budgets to regulate accesses.

The health monitor is split between the two domains. The first component is placed in the GPOS domain to observe the correct execution of both the OS and the application, and sends updates to the critical domain through dedicated channels in the communication layer provided by the hypervisor. The second element is located in the RTOS domain alongside the critical application to validate the information received, the freshness of incoming data, and their correctness. Such a monitor is useful to check whether the received input values are safe to be used, otherwise they will be replaced with others produced in the critical domain by a simpler but more robust algorithm.

The proposed architecture can be specialized according to the requirements of the system in which it has to be used. After surveying today's technologies with the purpose of developing autonomous systems with limited energy budgets, the architecture was specialized by adopting the following components:

- ZCU104 board by Xilinx/AMD, equipped with an Ultrascale+ MPSoC (XCZU7EV);
- Xilinx/AMD DPU accelerator (DPUCZDX8G) to be deployed onto the FPGA fabric of the Ultrascale+;
- CLARE-Hypervisor by (Accelerat: The CLARE Software Stack);
- Linux operating system for the non-critical, high-performance domain; and
- FreeRTOS as the real-time operating system for the critical domain.

The ZCU104, although conceived as a development board, allows matching SWaP-C constraints for several target applications, at least in their prototype stage. At the same time, the amount of FPGA resources available in the MPSoC installed on the ZCU104 allows deploying peripherals that are missing in the board, with a significant speed-up and flexibility in the hardware setup. Finally, the Deep Learning Processor Unit (DPU) by Xilinx is notably the most mature solution to date to accelerate AI algorithms using FPGA technology, relieving the designer from synthesizing DNN-specific accelerators and enabling the acceleration of multiple DNNs with the same FPGA design.

CLARE-Hypervisor (Accelerat: The CLARE Software Stack) follows a static approach with offline configurations and optimizations to allocate the platform resources to domains. Furthermore, it provides unique isolation and security features that make it an excellent choice for developing mixed-criticality CPS applications. It has been designed to explicitly support modern heterogeneous platforms, such as GPGPU- and FPGA-based MPSoC, to safely and securely control their

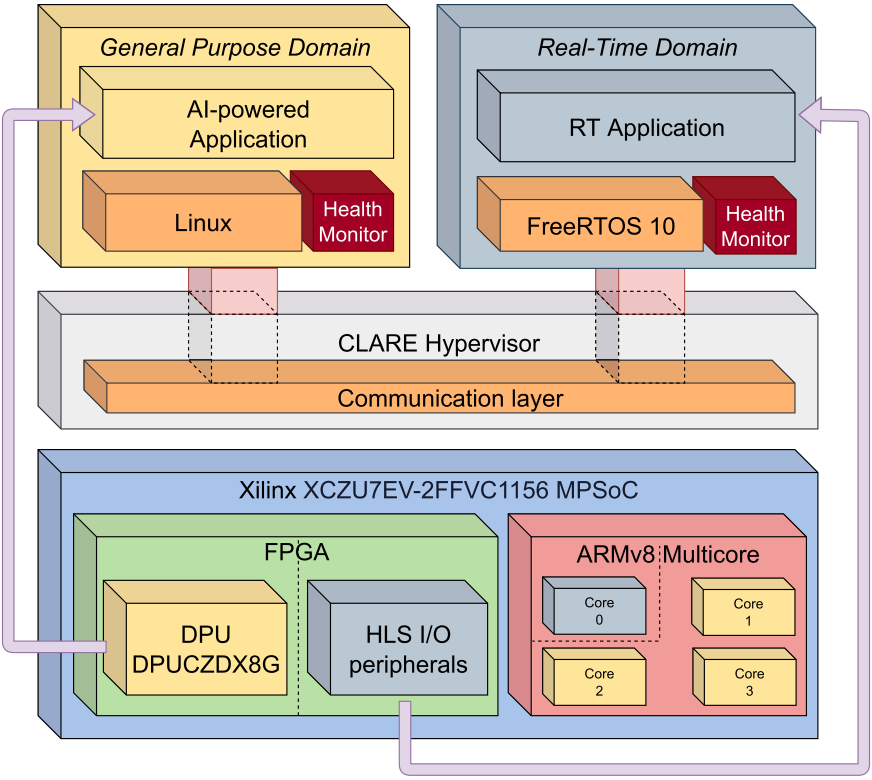


Fig. 1 Illustration of the proposed specialized architecture

computational resources. CLARE-Hypervisor also provides multi-domain virtualization of the FPGA area, enabling strong isolation also for PL components such as hardware accelerators.

Linux has been selected as GPOS for its extensive support for peripherals drivers, communication stacks, and modern AI frameworks.

FreeRTOS has been selected as RTOS because its execution model is suitable for timing analysis and, because of its diffusion, it includes a rich set of drivers for low-level devices.

The resulting specialized architecture is illustrated in Fig. 1.

4 The case for autonomous drones

This section describes how the architecture presented in Sect. 3 can be used to implement an AI-powered visual tracking application on a quadcopter drone equipped with an inertial measurement unit (IMU), a camera for object tracking, and two directional LiDAR sensors for obstacle detection, one pointing forward and one backward.

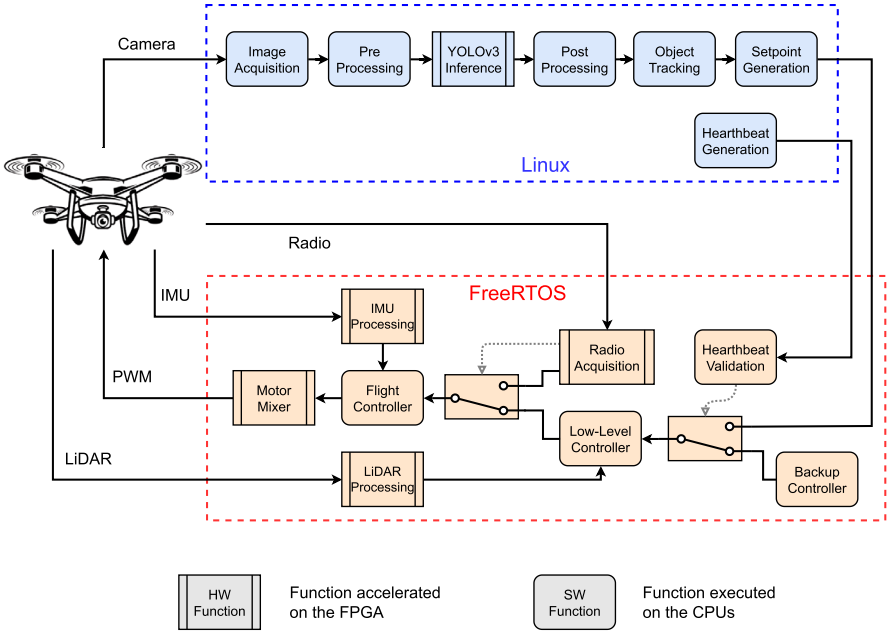


Fig. 2 Function diagram of the multi-domain application that controls the drone

The overall block diagram of the multi-domain application that controls the drone is illustrated in Fig. 2, which also distinguishes the functions executed in the Linux domain (blue blocks) from those running in the critical domain (orange blocks). In particular, the ARMv8 processing system is divided across domains, so that three out of four cores are assigned to the Linux domain, while the remaining one is assigned to the freeRTOS domain. The figure also highlights with a double border the modules that are either entirely implemented in FPGA or leverage the FPGA to accelerate some functions.

The main task of the Linux domain is the inference of a deep neural network (DNN) for real-time multiple object tracking using a strategy derived from DeepSORT (Wojke et al. 2017) and BYTEtrack (Zhang et al. 2021). The generated bounding boxes, paired with a unique ID of the object, are used to compute a set-point for the low-level drone controller running in the critical domain. In this context, support for hardware acceleration is essential to achieve acceptable performance, because all state-of-the-art neural trackers generate a significant workload that has to be executed in real-time (normally at the camera frame rate). Table 1 summarizes the main functions that compose the system.

4.1 Devices synthesized on the FPGA

The FPGA is used to synthesize a number of devices that are assigned to the virtual machines by the hypervisor. In particular, each device is exclusively assigned

Table 1 Application functions

Function Name	Domain	FPGA support	Description
Image acquisition	Linux	No	It captures a frame from the camera
Pre-processing	Linux	No	It adapts the image to the size required by YOLOv3 for the inference
YOLOv3 Inference	Linux	Yes	It uses the DPU to accelerate YOLOv3 runtime
Post-processing	Linux	No	It computes the bounding boxes of the detected objects
Object tracking	Linux	No	It solves the assignment problem and manages tracklets
Setpoint generation	Linux	No	It computes the position setpoint for the low-level controller
Heartbeat generation	Linux	No	It sends a control signal notifying the health of Linux
Heartbeat validation	freeRTOS	No	It checks the heartbeat signal to detect Linux faults
Backup controller	freeRTOS	No	It performs a backup routine
LiDAR processing	freeRTOS	Yes	It reads the LiDARs and sends data to the low-level controller
Low-level controller	freeRTOS	No	It keeps the drone at a given safety distance from a possible obstacle
Radio acquisition	freeRTOS	Yes	It reads the radio data from the PPM decoder
IMU processing	freeRTOS	Yes	It reads the sensors and computes angular rates and attitudes
Flight controller	freeRTOS	No	It performs a PID control to stabilize the drone
Motor mixer	freeRTOS	Yes	It computes the motor data from the trust provided by the flight controller

in a pass-through way to one domain only, while the hypervisor is responsible for providing strong isolation. In particular, the Xilinx DPU device is accessible by the Linux domain, while all the other devices are assigned to the critical domain. All custom devices synthesized on the programmable logic are described in the following list:

1. *DPU*: The Deep Learning Processor Unit (DPU) is a softcore provided by Xilinx to efficiently accelerate the inference of deep neural networks.
2. *Radio decoder*: It takes the pulse position modulated (PPM) signal from the radio receiver, decodes it, and puts the corresponding digital values in a set of registers. Without the help of specialized hardware, PPM signals would have to be managed in software using, for instance, GPIOs configured to raise interrupts at each edge in the signal to process it. This may easily lead to poor performance and excessive interference on the processors due to the service of interrupts and the consequent context switches. The use of a dedicated FPGA component to handle the PPM signal of the radio receiver hence relieves the processors from this burden and reduces the corresponding overhead and jitter.

3. *I^2C device*: The ZCU104 board allows exposing an I^2C peripheral working with 1.8 V logic levels, while the adopted IMU works with 3.3 V logic levels. To avoid introducing third-party electronics to adapt the logic levels (e.g., using a voltage-level translator), an AXI-based 3.3 V I^2C master device to be deployed on FPGA was developed.
4. *UART device*: A custom AXI-based UART peripheral to be deployed on FPGA was developed for the same reasons mentioned above, given that the adopted LiDAR works with 3.3 V logic levels.
5. *PWM driver*: It is used to generate pulse width modulation (PWM) signals to drive the drone motors. Although the Ultrascale+ MPSoC allows generating PWM signals by means of triple timer counters (TTC), a specialized FPGA module was developed for the sake of simplicity and flexibility.

Efficient implementations of the drivers for the above peripherals, except the DPU, were performed from scratch to offload the CPU as much as possible, as well as minimize execution time variability and the number of memory accesses.

4.2 Inter-domain communication channels

The two domains exchange data by means of two non-blocking communication channels based on shared-memory regions provided by CLARE, where the Linux domain acts as a producer and the critical one as a consumer. The channels are accessed by means of a middleware (available for both Linux and FreeRTOS) that does not require the intervention of the hypervisor at each access and ensures wait-free synchronization in the presence of concurrent accesses. The first channel is used to exchange setpoints for the drone controller, whereas the second one is used to transmit heartbeat packets for health monitoring.

4.3 Linux domain

The Linux domain is responsible for visual tracking and navigation. It includes four tasks, namely `Camera`, `Detector`, `Tracker`, and `HB generator`. Details on these tasks are reported in Table 2.

The `Camera` task periodically captures a new frame from the camera and puts it in a queue of frames ready to be processed. The `Detector` task performs object detection by accelerating the inference of a YOLOv3 (Redmon and Farhadi 2018) deep neural network on the Xilinx DPU. The YOLOv3 model was trained on the cityscape dataset (Cordts et al. 2015), using a Darknet-53 backbone, modified (with respect to the standard implementation) to process an extra output from the pyramidal feature extraction stage, to improve its performance.

To be executed on the DPU, the neural model was *quantized* (transforming its weights from 32-bit floating point values to 8-bit integer values) and *pruned* (removing the parameters with less contribution) to reduce the memory footprint and the amount of operations to be executed by the accelerator. The model was then

Table 2 Linux application-level task set. Priority value ranges between 1 and 99, where higher values correspond to a higher priority

Task Name	Period (ms)	Priority	Called functions
`Camera`	34	14	Image acquisition
`Detector`	36	10	Pre-processing YOLOv3 inference Post-processing
`Tracker`	34	12	Object tracking Setpoint generation
`HB generator`	4	16	Heartbeat generation

compiled to generate DPU-specific instructions to speed up the inference. This process was performed by means of Xilinx's Vitis AI.

Besides the inference, a YOLOv3 model requires a pre-processing and a post-processing stage. In particular, the pre-processing stage takes a frame from the Ready-frame queue, rescaling and normalizing it to the YOLOv3 input size (512× 256×3). The post-processing stage derives all bounding-box coordinates, executes the Non-Maximum Suppression algorithm, and inserts the result into a queue to be processed by the `Tracker` task. The purpose of the Object Tracking function is to predict (by a Kalman filter) the position of a tracked object to keep identifying it even when it is missed by the object detector. The coordinates of the tracked target are then used to compute the setpoint to be sent to the critical domain.

The minimum, average, and maximum observed execution times of the functions involved in the object detection pipeline are reported in Table 3.

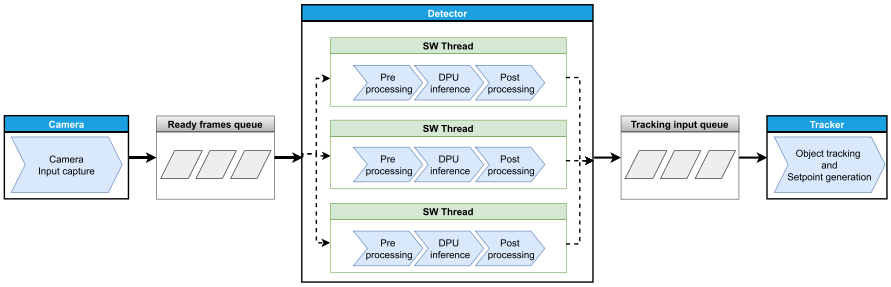
The execution times reported in Table 3 have been measured by a performance counter for a time interval of 20 min, running each task in isolation to prevent interference from higher priority tasks. As clear from the results, the pre-processing and post-processing functions represent the major bottleneck in achieving real-time performance. To overcome this problem, all three functions performing object detection were implemented as a concurrent multi-thread pipeline, as illustrated in Fig. 3.

The number of parallel threads has been set equal to the number of cores assigned to the Linux domain (three, in our setup), with the benefit of increasing the throughput from 16 to 27 frames per second (FPS). Note that the tracking function cannot be parallelized, because the position of the tracked objects depends on the computations related to the previous frame.

The `HB generator` task is responsible for the health monitoring activities within the Linux domain. It periodically produces two heartbeat timestamps, reporting the health of the Linux system and the application. The system-level timestamp is updated each time this task is executed, while the application-level timestamp is updated every time a new setpoint is generated by the `Tracker` task.

Table 3 Execution times of the functions involved in the object detection pipeline

Function Name	Min (ms)	Avg (ms)	Max (ms)
Image pre-processing	24.47	25.48	31.84
YOLOv3 DPU inference	8.52	8.93	9.36
Post-processing	27.43	30.69	35.53



Priority value ranges between 0 and 99. Higher values correspond to a higher priority.

Fig. 3 Diagram of the multi-thread pipeline used for object detection and tracking

4.4 Critical (FreeRTOS) domain

The critical domain is composed of four tasks, whose periods and priorities are reported in Table 4, while execution times are reported in Table 5.

Observed execution times of Table 5 were obtained using the same technique adopted for Table 3.

The `HB checker` task periodically verifies the heartbeat values sent by the `HB generator` to detect possible faults related to the Linux domain, notifying the `Safety module` task when a fault is detected. The reaction time to a fault is given by the period of this task multiplied by a user-defined parameter that indicates the number of tolerable task executions without heartbeat updates.

The `Backup controller` task is responsible for handling an alternative setpoint generation when the fail-safe mode is activated. As shown in Table 5, the execution time of this task is significantly shorter than the others because, in the current implementation, it simply keeps the quadcopter hovering in the position recorded when the fault is detected. In general, however, this task could be used to implement more complex actions, such as controlling the drone to perform a safe landing.

The `Safety module` task is responsible for producing the input to the flight controller. If no fault is detected by the `HB checker` task, it reads the latest setpoint provided by the Linux application, otherwise, it switches to the `Backup controller` as an alternative source of setpoints. The `Safety module` task also performs high-level health monitoring functions related to the behavior of the application, verifying that the provided set points are not jeopardizing designated safety constraints. The current implementation reads LiDAR data and conducts an additional control loop to keep the drone at a minimum user-defined distance from possible obstacles on the path to the next waypoint.

The `Flight Control` task can work in two different modes, manual and AI-driven, selected by a switch on the radio transmitter. In manual mode, the setpoint is taken from the radio transmitter, while in AI-driven mode the setpoint is taken from the `Safety module` task. In both cases, the `Flight Control` task reads the IMU data and stabilizes the quadcopter using two hierarchical PID control loops.

Table 4 FreeRTOS task set. Priority value ranges between 0 and 99, where higher values correspond to a higher priority

Task Name	Period (ms)	Priority	Called functions
`HB checker`	4	4	Heartbeat validation
`Safety module`	10	1	LiDAR processing Low-Level controller Setpoint switch
`Flight control`	4	2	Flight controller IMU processing Radio acquisition Motor mixer
`Backup Controller`	4	3	Backup Controller

Table 5 Execution times of the FreeRTOS tasks

Task Name	Min (μs)	Avg (μs)	Max (μs)
`HB checker`	10	11	35
`Safety module`	338	349	351
`Flight control`	437	438	450
`Backup Controller`	0.062	0.08	0.126

The inner loop controls the angular rates, while the outer loop controls the quadcopter attitude. Finally, the outputs generated by the Flight controller are sent to the Motor mixer for actuation.

5 Experimental results

This section reports some experiments aimed at showing how the proposed multi-domain architecture can effectively be implemented with a negligible impact on the overall performance, with respect to an implementation without a multi-domain design.

5.1 Experimental setup

The vehicle used for the implementation is an F450 class quadcopter with four 1045 propellers controlled in the X standard configuration. Figure 4 shows the block diagram of the Xilinx ZCU-104 Ultrascale+ MPSoC configuration used for our application.

The camera is a Logitech HD Pro C920, which natively provides frames with a resolution of 320×240, which is the closest one to the input size of the adopted neural network. The IMU is an MPU-9250, a 9-DOF inertial device with gyroscopes, accelerometers, and magnetometers that communicates with the system using a 3.3 V I^2C device (set in *Fast mode* to work with 400 kHz clock) implemented inside the

FPGA programmable logic. Additionally, this device has a configurable filter that can be used to reduce noise and improve communication reliability.

The radio receiver also takes advantage of FPGA acceleration. It uses the custom AXI peripheral we designed to handle the PPM external interrupts (EXTI line) to offload the CPU from the interrupt service routine and reduce interferences and jitter on the control task.

LiDARs communicate through a dedicated UART implemented in FPGA with a baud rate of 115,200 bit/s. Although it is possible to generate PWM signals using the hardware timers provided by the processing system, we built our custom PWM peripheral to be completely independent from the CPU, ensuring exclusive access by the RTOS domain using a dedicated driver.

The Electronic Speed Controllers (ESCs) used in the quadcopter are the BL-Heli, which accept PWM pulses from 50Hz (legacy PWM) to 12KHz (OneShot-42 protocol), so they can perfectly handle the 250Hz PWM signal of the custom HLS peripheral. The Racestar 980 KV brushless motors are paired with the 1045 propellers and can provide a thrust of 3.2 kg.

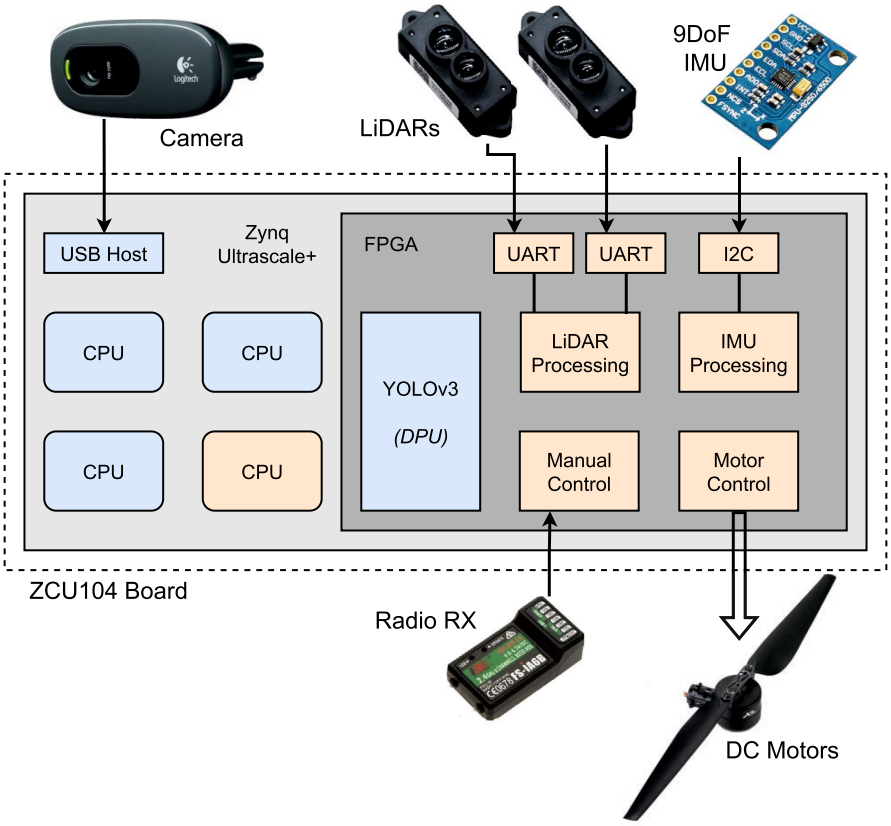


Fig. 4 Hardware platform application diagram

The Xilinx DPU is the most complex and resource-demanding FPGA device required by the application because it uses a very large portion of LUTs in the programmable logic region and almost all the marginal resources of the board, like Digital Signal Processors (DSP), Block RAM (BRAM), and Ultra RAM (URAM), as reported in Table 6.

Note that the design occupies less than half of the available resources. This choice is motivated by the fact that, in the next future, the complete application will be moved to the Kria K26 SOM, with an ad-hoc carrier board to further reduce weight and consumption, but its SoC contains half of the resources available on the XCZU7EV-2FFVC1156 powering the ZCU104 board.

This specific setup led to the power specifications reported in Fig. 5. How it can be seen from the figure, the major source of power consumption resulted to be the dynamic one in PL due to the DPU runtime and the intense switching activity it involves.

The battery selected as a power source for the system is a 3300 mAh 4 S LiPo, which is sufficient to guarantee the maximum power consumption specified in Fig. 5. In fact, it can always provide a sufficient voltage/current to the D36V50F12 voltage regulator to maintain it in its optimal operating spot and a stable power rail of 12 V with a maximum load current of 4.5 A. The maximum payload of the quadcopter resulted be 1.5 kg in this specific configuration. Additionally, the external devices connected to the platform present the following power specifications. The Racestar BR2212 motor datasheet reports that, applying a Voltage of 11.1 V, each motor paired with 1045 propellers (the ones we used) absorbs a maximum current of 10.6 A, for a total maximum current of $10.6 \times 4 = 42.4$ A. The Fs-iA6 radio receiver, supplied with a voltage between 4 and 6.5 V, has a maximum power consumption no higher than 520 mAh. The power consumption of the MPU-9250 is quite low, since it must be supplied with a voltage of 3.3 V and reaches 3.7 mA when all three sensors (gyroscope, accelerometer, and magnetometer) are powered on. Each directional LiDAR works with an input voltage between 3.7 V and 5.2 V, absorbs an average current less or equal to 70 mA, a peak current of 150 mA, and a total power of less than 0.35 W. In this setup, we used two of these sensors.

Table 6 Utilization of FPGA resources

Resource	Utilization	Available	Utilization %
LUT	50966	230400	22.12
LUTRAM	5741	101760	5.64
FF	102121	460800	22.16
BRAM	107	312	34.29
URAM	40	96	41.67
DSP	690	1728	39.93
IO	22	360	6.11
BUFG	4	544	0.74
PLL	1	16	6.25

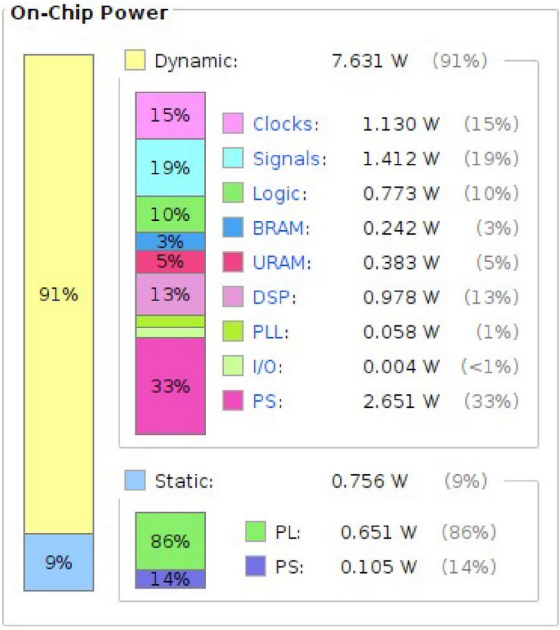


Fig. 5 Xilinx Ultrascale+ MPSoC power report

5.2 Object detection

In this experiment, the execution behavior of the `Detector` task has been analyzed both on the multi-domain system and the Petalinux default flow, to evaluate the impact of the hypervisor. For the sake of fairness, since the multi-domain implementation allocates three out of four available cores to Linux, the measures in the configuration without hypervisor have been performed by turning one core off for the Linux application. Furthermore, in the multi-domain implementation, the core running FreeRTOS was programmed not to generate traffic on the bus to avoid extra memory access conflicts, which would cause a performance degradation not due to the hypervisor.

Figure 6 shows the frame rate distribution of the object detector task under the configuration without hypervisor (red bars) and with hypervisor (blue bars). The part of the plot in purple color represents the overlapping portion of the two histograms. The smoother envelopes of the two histograms are also shown with the corresponding colors. As clear from the plot, the two histograms almost overlap, meaning that the hypervisor introduces negligible overhead and does not degrade the overall system performance significantly. On the other hand, the hypervisor allows isolating the two execution domains, preventing malicious attacks carried out on the Linux domain to propagate and affect the critical functions running in the RTOS domain.

In another test, the object detection performance resulting from a 3-core configuration with hypervisor has been compared with the one achievable on the full ZCU-104 without hypervisor, that is, assigning all four cores to Linux. The results are

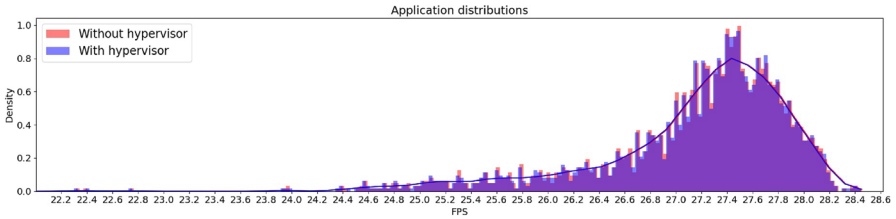


Fig. 6 Frame rate distributions for the `Detector` task on a 3-core configuration without hypervisor (red bars) and with hypervisor (blue bars)

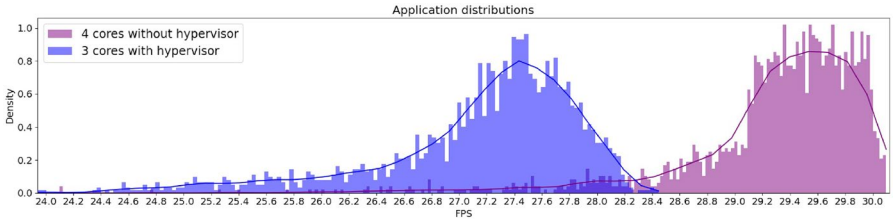


Fig. 7 Frame rate distributions for the `Detector` task on a 3-core configuration with hypervisor (blue bars) and on a 4-core configuration without hypervisor (red bars)

illustrated in Fig. 7, which shows that, by allocating one extra core to Linux, the average frame rate of the object detection task increases from 27.5 FPS to 29.6 FPS.

Note that, in the 4-core configuration, the object detection pipeline uses four parallel threads to match the number of physical cores available on the platform. As expected, this leads to a performance increase, but the observed improvement is not significant, since the processing pipeline is constrained by the acquisition rate of the camera (30 FPS), which limits the benefit of the increased HW and SW parallelism.

Viewed from another angle, the results reported in Fig. 7 show that the two-domain architecture enabled by the hypervisor does not significantly degrade the performance, with respect to a full platform configuration, but certainly provides other relevant advantages in terms of time predictability and security for the critical components of the system.

5.3 Application-level end-to-end delays

This section reports on two experiments aimed at showing the time it takes for the generated data to be propagated from one domain to the other. Since this operation involves a data exchange between two very different operating systems, the communication latency depends on different factors. In particular, from the moment in which the data is written by Linux into the shared memory, three factors come into play: (i) the period of the consumer task running in the FreeRTOS domain, (ii) the time at which this task is scheduled by FreeRTOS, and (iii) the interference experienced by the consumer from the other tasks, which depends on the assigned priorities and the task execution times.

Figure 8 illustrates a possible interleaving of the producer and consumer tasks, where the delay is significant. In the figure, the message is sent by the producer at time t_1, it is delivered to the other domain at time t_a, and finally consumed at time t_2. As it can be seen, the overall end-to-end delay ($t_2 - t_1$) is given by the sum of the channel latency (L_c), the activation interval (A_c), the interference of the high-priority tasks (I_{hp}), and the computation time of the consumer task (C_c). Since the channel latency is always below one microsecond and the execution times of FreeRTOS tasks are in the order of a few hundred microseconds, the major contribution to the end-to-end communication delay is due to the period of the consumer task, which is in the order of milliseconds.

The best-case situation for the end-to-end communication delay is illustrated in Fig. 9, where the consumer task is executed just after the message is delivered to the FreeRTOS domain. In this case, the end-to-end delay is in the order of a few hundred microseconds.

The end-to-end delay measurements performed in this experiment confirm the observations reported above. Figure 10 shows the distribution of the end-to-end delay for the setpoint communication, measured over one hour of continuous execution from the time the setpoint is generated in Linux to the time it is read in FreeRTOS by the `Safety module` task, which has a period of 10 ms and the lowest priority.

As expected, the maximum observed delay resulted to be 9.5 ms (close to the period of 10 ms assigned to the `Safety module` task), whereas the minimum observed delay resulted to be 890 μs, due to the sum of its own execution time and the suffered interference from some higher priority task.

Figure 11 shows the distribution of the end-to-end delay from the time the heartbeat is generated in Linux to the time it is received in FreeRTOS.

In this case, the maximum observed delay resulted to be of 3.74 ms (close to the period of 4 ms of the `HB checker` task), whereas the minimum observed delay resulted to be of about 303 μs, shorter than the other, because the `HB checker` task has the highest priority and cannot suffer interference from the other tasks.

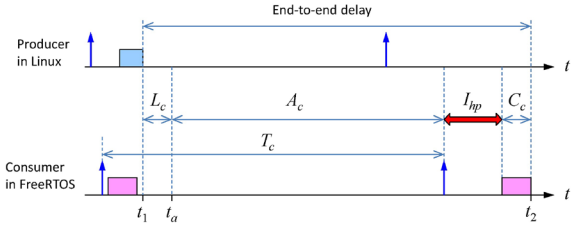


Fig. 8 Example of task interleaving characterized by a long end-to-end delay. In this case, the delay is mainly dominated by the activation interval A_c, which has the same order of magnitude of the consumer period T_c (ms), whereas the channel latency L_c, the interference I_{hp} from the high-priority tasks, and the consumer computation time C_c are at least two orders of magnitude smaller

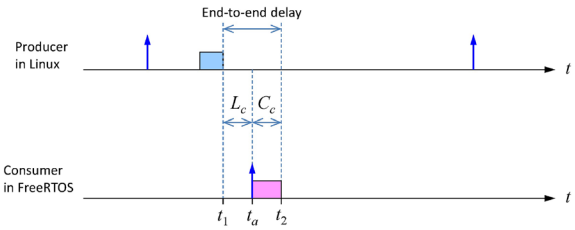


Fig. 9 Example of task interleaving characterized by a short end-to-end delay. It is mainly caused by the channel latency L_c and the execution time C_c of the consumer task

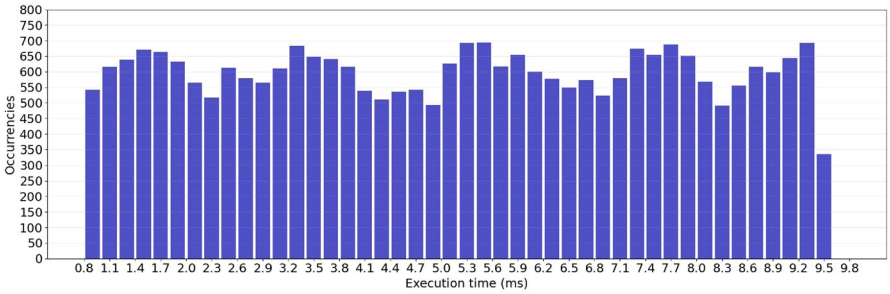


Fig. 10 Setpoint transmission delay

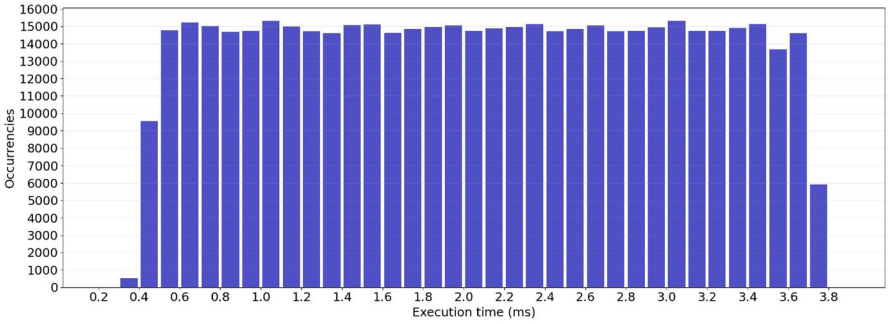


Fig. 11 Heartbeat transmission delay

5.4 Fault reaction time

A final experiment was carried out to measure the latency of the fail-safe procedure triggered by a system fault. For this specific test, the drone is programmed to track a target by controlling only the yaw angle. Hence, Fig. 12 reports the variation of the yaw angle during a tracking operation, when the backup controller is invoked to keep the drone in a safe state after a system fault is injected in Linux.

In this experiment, the system heartbeat validation threshold was set to 3, meaning that the `HB checker` task can tolerate 3 readings of heartbeat data in the FreeRTOS domain without detecting an update. Note that, since the period of the `HB`

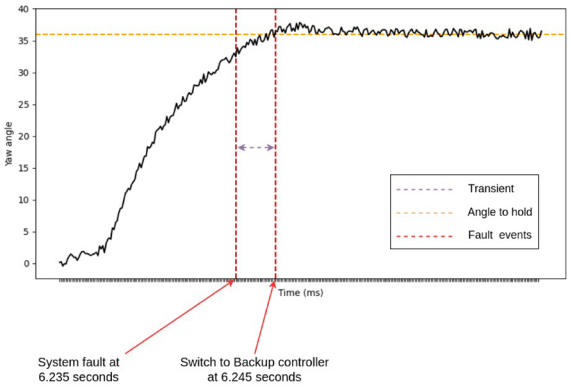


Fig. 12 Reaction time for the Backup controller to a system fault

`checker` is set to 4 ms and the threshold is 3, the expected delay to detect a fault is between 8 and 12 ms. In Fig. 12, the measured delay is represented by the purple arrow between the two vertical red dashed lines, denoting the transient interval between the normal functioning and the fail-safe mode.

In this test, the fault has been injected in Linux after 6.235 s from the beginning of the plot and the backup controller took place at $t = 6.245$ seconds, that is, after about 10 ms. The orange horizontal line highlights the yaw angle recorded when the fail-safe mode was enabled. In this implementation, the backup controller has been programmed to keep the yaw angle to that reference value.

The plot shows that, when the backup controller was activated, the yaw angle was 36.3° and the flight controller acted to keep the yaw angle at this reference value, while simultaneously keeping both pitch and roll angles at 0° (hovering). Notice that a little overshoot occurs in most cases, since the yaw PID controller is the one with less authority over the physical system. In fact, yaw actuation is generated by differences in motor torques, while pitch and roll are generated by changing thrusts, so the faster the control (larger gains) the higher the overshoot to be compensated. This can be observed in the figure after the second red line, when the angle continues increasing for a few milliseconds, after which it converges to the reference angle recorded at the fail-safe activation. This behavior is caused by the system dynamics rather than by a delay in the execution of the flight control task.

Note that, without the fail-safe mechanism, the drone would no longer receive a setpoint and therefore would continue to apply the last valid setpoint received, thus keeping rotating around itself. In a more complex use case involving multiple degrees of freedom, this situation would inevitably lead to dangerous consequences.

6 Conclusions

This paper presented a fully functional implementation of a multi-domain architecture for running software components of different criticality and performance requirements on a single heterogeneous platform. The architecture has been implemented on a Xilinx ZCU104 MPSoc to perform AI-based visual tracking on a drone.

The CLARE-Hypervisor (Accelerat: The CLARE Software Stack) has been used to realize two isolated execution domains: one powered by Linux, responsible for image acquisition, object detection, and AI-based tracking, and one powered by FreeRTOS, responsible for running more critical tasks as the low-level control of the drone.

The results presented in Sect. 5 confirm that the proposed approach is effective to achieve both high timing predictability, needed for guaranteeing a real-time performance, as well as a contained power consumption, essential to save energy in battery operated cyber-physical systems. An example of hypervisor-assisted health monitoring was demonstrated in the context of the studied application to trigger fail-safe routines to bring the drone in a safe state when a fault is detected.

To evaluate the improvement of the proposed solution in more complex scenarios, the ZCU104 board will be replaced with a Kria K26 SOM paired with an ad-hoc carrier board to better fit the SWaP-C requirements of drone systems.

Acknowledgements This work has been partially supported by the Italian Ministry of University and Research (MUR) under the SPHERE project funded within the PRIN-2017 framework (Grant No. 93008800505) and the VIRMA project funded by the EU PNRR plan with DM-737/2021.

Funding Open access funding provided by Scuola Superiore Sant'Anna within the CRUI-CARE Agreement.

References

Accelerat: The CLARE Software Stack. https://accelerat.eu/clare

Almeida J, Prochazka M (2009) Safe and Secure Partitioning with Pikeos: Towards Integrated Modular Avionics in Space. In: Proceedings of DASIA 2009 Data Systems in Aerospace, p. 27

AMD Xilinx: DPU - Deep Learning Processing Unit. https://www.xilinx.com/products/intellectual-property/dpu.html

AMD Xilinx: FINN Framework. https://xilinx.github.io/finn/

AMD Xilinx: Vitis AI - Adaptable and Real-Time AI Inference Acceleration. https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html

Biondi A, Nesti F, Cicero G, Casini D, Buttazzo G (2020) A Safe, Secure, and Predictable Software Architecture for Deep Learning in Safety-Critical Systems. IEEE Embedded Systems Letters 12(3):78–82

Biondi A, Casini D, Cicero G, Borgioli N, Buttazzo G, Patti G, Leonardi L, Bello LL, Solieri M, Burgio P, Olmedo IS, Ruocco A, Palazzi L, Bertogna M, Cilardo A, Mazzocca N, Mazzeo A (2021) Sphere: a multi-soc architecture for next-generation cyber-physical systems based on heterogeneous platforms. IEEE Access 9:75446–75459

Biondi A, Balsini A, Pagani M, Rossi E, Marinoni M, Buttazzo G (2016) A Framework for Supporting Real-Time Applications on Dynamic Reconfigurable FPGAs. In: Proc. of the IEEE Real-Time Systems Symposium (RTSS 2016), Porto, Portugal

Cavicchioli R, Capodieci N, Bertogna M (2017) Memory Interference Characterization Between CPU Cores and Integrated GPUs in Mixed-Criticality Platforms. In: Proc. of the 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2017)

Cordts M, Omran M, Ramos S, Scharwächter T, Enzweiler M, Benenson R, Franke U, Roth S, Schiele B (2015) The Cityscapes Dataset. In: CVPR Workshop on The Future of Datasets in Vision

Craveiro J, Rufino J, Schoofs T, Windsor J (2009) Flexible Operating System Integration in Partitioned Aerospace Systems. In: Actas do INForum - Simposio de Informatica, p 49–60

Crespo A, Balbastre P, Simó J, Coronel J, Pérez DG, Bonnot P (2018) Hypervisor-based multicore feedback control of mixed-criticality systems. IEEE Access 6:50627–50640

Crespo A, Ripoll I, Masmano M, Arberet P, Jean-Jacques M (2009) XtratuM: an Open Source Hypervisor for TSP Embedded Systems in Aerospace. In: Proceedings of DASIA 2009 Data Systems in Aerospace

Crespo A, Ripoll I, Masmano M, Arberet P, Metge JJ (2009) XtratuM: An open source hypervisor for TSP embedded systems in aerospace. In: Data Systems In Aerospace (DASIA), Istanbul, Turkey (May 26-29)

CubePilot: The Cube Autopilot. https://www.cubepilot.com/

Ding C, Wang S, Liu N, Xu K, Wang Y, Liang Y (2019) REQ-YOLO: A Resource-Aware, Efficient Quantization Framework for Object Detection on FPGAs. arXiv:1909.13396

Fahim F, Hawks B, Herwig C, Hirschauer J, Jindariani S, Tran N, Carloni LP, Di Guglielmo G, Harris P, Krupa J, Rankin D, Valentin MB, Hester J, Luo Y, Mamish J, Orgrenci-Memik S, Aarrestad T, Javed H, Loncar V, Pierini M, Pol AA, Summers S, Duarte J, Hauck S, Hsu S-C, Ngadiuba J, Liu M, Hoang D, Kreinar E, Wu Z (2021) hls4ml: an Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices. arXiv. arXiv:2103.05579

Farrukh A, West R (2022) FLYOS: Integrated Modular Avionics for Autonomous Multicopters. In: 28th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2022)

Gaska T, Werner B, Flagg D (2011) Applying virtualization to avionics systems - the integration challenges. IEEE Aerospace and Electronic Systems Magazine 26

Gholami A, Kim S, Zhen D, Yao Z, Mahoney M, Keutzer K (2022) A Survey of Quantization Methods for Efficient Neural Network Inference. Low-Power Computer Vision, p 291–326

Gutiérrez CSV, Juan LUS, Ugarte IZ, Vilches VM (2018) Towards a distributed and real-time framework for robots: Evaluation of ROS 2.0 communications for real-time robotic applications. arXiv:1809.02595

Happe M, Traber A, Keller A (2015) Preemptive hardware multitasking in reconos. In: International Workshop on Applied Reconfigurable Computing

Intel Corporation: Overview of Intel Ready to Fly Drone. https://www.intel.it/content/www/it/it/support/articles/000023272/drones/development-drones.html

Ji Q, Dai C, Hou C, Li X (2021) Real-time embedded object detection and tracking system in zynq soc. EURASIP Journal on Image and Video Processing 21

Klein G, Andronick J, Fernandez M, Kuz I, Murray T, Heiser G (2018) Formally Verified Software in the Real World. Communications of the ACM 61(10):68–77

Leiner B, Schlager M, Obermaisser R, Huber B (2007) A Comparison of Partitioning Operating Systems for Integrated Systems. In: Computer Safety, Reliability, and Security. Springer, Berlin p 342–355

Lelli J, Scordino C, Abeni L, Faggioli D (2016) Deadline scheduling in the Linux kernel. Software: Pract Exper 46(6):821–839

Liang T, Glossner J, Wang L, Shi S, Zhang X (2021) Pruning and quantization for deep neural network acceleration: a survey. Neurocomputing 461:370–403

Liu M, Niu J, Wang X (2017) An autopilot system based on ros distributed architecture and deep learning. In: 2017 IEEE 15th International Conference on Industrial Informatics (INDIN), pp 1229–1234

LYNX Software Technologies: LynxSecure Embedded Hypervisor and Separation Kernel. http://www.lynx.com/products/hypervisors/

Meier L, Honegger D, Pollefeys M (2015) Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In: 2015 IEEE International Conference on Robotics and Automation (ICRA), p 6235–6240

Musliner D, Hendler J, Agrawala A, Durfee E, Strosnider J, Paul CJ (1995) Challenges of real-time ai. Computer 28:58–66

O'Kelly M, Sukhil V, Abbas H, Harkins J, Kao C, Pant YV, Mangharam R, Agarwal D, Behl M, Burgio P, Bertogna M (2019) F1/10: An Open-Source Autonomous Cyber-Physical Platform. arXiv. arXiv:1901.08567

Pérez H, Gutiérrez JJ (2016) Handling heterogeneous partitioned systems through ARINC-653 and DDS. Comput. Stand. Interfaces 50:258-268
Pérez H, Gutiérrez JJ, Peiró S, Crespo A (2016) Distributed architecture for developing mixed-criticality systems in multi-core platforms. J Syst Soft 123:145–159
Qasaimeh M, Denolf K, Lo J, Vissers K, Zambreno J, Jones PH (2019) Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels. In: 2019 IEEE International Conference on Embedded Software and Systems (ICESS), p 1–8
Radanliev P, De Roure D, Van Kleek M, Santos O, Ani U (2020) Artificial intelligence in cyber physical systems. AI & Sociaty 36:783–796
Redmon J, Farhadi A (2018) YOLOv3: An Incremental Improvement. arXiv
Reke M, Peter D, Schulte-Tigges J, Schiffer S, Ferrein A, Walter T, Matheis D (2020) A Self-Driving Car Architecture in ROS2, p 1–6
Rupnow K, Fu W, Compton K (2009) Block, drop or roll(back): alternative preemption methods for rh multi-tasking. In: 2009 17th IEEE Symposium on Field Programmable Custom Computing Machines, p 63–70
Sciangula G, Restuccia F, Biondi A, Buttazzo G (2022) Hardware Acceleration of Deep Neural Networks for Autonomous Driving on FPGA-based SoC. In: the 25th Euromicro Conference on Digital System Design (DSD), Maspalomas, Gran Canaria, Spain
Scordino C, Savino IM, Cuomo L, Miccio L, Tagliavini A, Bertogna M, Solieri M (2020) Real-time virtualization for industrial automation. In: 2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), vol. 1, p 353–360
Seng KP, Lee PJ, Ang LM (2021) Embedded intelligence on fpga: survey, applications and challenges. Electronics 10(8):895
Seyoum B, Pagani M, Biondi A, Balleri S, Buttazzo G (2021) Spatio-temporal optimization of deep neural networks for reconfigurable FPGA SoCs. IEEE Transactions on Computers 70(11):1988–2000
SYSGO: PikeOS Hypervisor. http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept
Vaishnav A, Pham KD, Koch D (2018) A survey on fpga virtualization. In: 2018 28th International Conference on Field Programmable Logic and Applications (FPL), pp 131–1317
Wang C, Luo Z (2022) A review of the optimal design of neural networks based on fpga. Appl Sci 12(21):10771
Wojke N, Bewley A, Paulus D (2017) Simple Online and Realtime Tracking with a Deep Association Metric. In 2017 IEEE international conference on image processing (ICIP), 3645–3649 Sep 2017
Zhang Y, Sun P, Jiang Y, Yu D, Weng F, Yuan Z, Luo P, Liu W, Wang X (2021) ByteTrack: multi-Object Tracking by Associating Every Detection Box. arXiv
Zhou S, Wang Y, Wen H, He Q, Zou Y (2017) Balanced quantization: an effective and efficient approach to quantized neural networks. Journal of Computer Science and Technology 32

Edoardo Cittadini received the master's degree (cum laude) in embedded computing systems engineering jointly offered by the Scuola Superiore Sant'Anna of Pisa, Pisa, Italy, and the University of Pisa, Pisa, in 2021. He is currently pursuing the Ph.D. degree with the Real-Time Systems (ReTiS) Laboratory, Scuola Superiore Sant'Anna of Pisa. His research interests include cyber-physical systems, artificial intelligence on heterogeneous platforms, real-time object tracking, and hypervisor technologies.

Mauro Marinoni is a research affiliate at the Scuola Superiore Sant'Anna of Pisa. He received a M.S. in Computer Engineering at the University of Pavia in 2003, where he also obtained a Ph.D. in Computer Engineering in 2007. He joined the ReTiS Lab since 2007, where he has been Assistant Professor from 2009 to 2020. He coordinated several European and industrial projects in different application fields, from e-Health devices to autonomous and distributed systems.

Alessandro Biondi is associate professor at the Real-Time Systems (ReTiS) Laboratory of the Scuola Superiore Sant'Anna. He graduated (cum laude) in Computer Engineering at the University of Pisa, Italy, within the excellence program, and received a Ph.D. in computer engineering at the Scuola Superiore Sant'Anna under the supervision of Prof. Giorgio Buttazzo and Prof. Marco Di Natale. In 2016, he has been visiting scholar at the Max Planck Institute for Software Systems (Germany). His research interests include design and implementation of realtime operating systems and hypervisors, schedulability analysis, cyber-physical systems, synchronization protocols, and safe and secure machine learning. He was recipient of seven Best Paper Awards, one Outstanding Paper Award, the ACM SIGBED Early Career Award 2019, and the EDAA Dissertation Award 2017.

Giorgiomaria Cicero is Senior Research Fellow at the Real-Time Systems (ReTiS) Laboratory of the Scuola Superiore Sant'Anna of Pisa and CEO at Accelerat Srl, a spin-off company of Scuola Superiore Sant'Anna. He graduated (cum laude) in Embedded Computing Systems Engineering, a joint Master degree by the Scuola Superiore Sant'Anna of Pisa and University of Pisa. He has been visiting trainee at the European Space Agency (ESTEC, Netherlands). His research interests include software predictability in multi-processor systems and heterogeneous platforms, system-level cyber-security hardening techniques, and design and implementation of real-time operating systems and hypervisors.

Giorgio Buttazzo is full professor of computer engineering at the Scuola Superiore Sant'Anna of Pisa. He graduated in Electronic Engineering at the University of Pisa, received a M.S. degree in Computer Science at the University of Pennsylvania, and a Ph.D. in Computer Engineering at the Scuola Superiore Sant'Anna of Pisa. He has been Editor-in-Chief of Real-Time Systems, Associate Editor of IEEE Transactions on Industrial Informatics and ACM Transactions on Cyber-Physical Systems. He is IEEE fellow since 2012, wrote 7 books on real-time systems and more than 300 papers in the field of real-time systems, robotics, and neural networks, receiving 15 best paper awards.