Real-Time Systems (2024) 60:150–196
https://doi.org/10.1007/s11241-023-09412-2

ChamelIoT: a tightly- and loosely-coupled hardware-assisted OS framework for low-end IoT devices

Miguel Silva[1] · **Tiago Gomes**[1] · **Mongkol Ekpanyapong**[2] · **Adriano Tavares**[1] · **Sandro Pinto**[1]

Accepted: 30 October 2023 / Published online: 20 December 2023

Abstract

The evergrowing Internet of Things (IoT) ecosystem continues to impose new requirements and constraints on every device. At the edge, low-end devices are getting pressured by increasing workloads and stricter timing deadlines while simultaneously are desired to minimize their power consumption, form factor, and memory footprint. Field-Programmable Gate Arrays (FPGAs) emerge as a possible solution for the increasing demands of the IoT. Reconfigurable IoT platforms enable the offloading of software tasks to hardware, enhancing their performance and determinism. This paper presents ChamelIoT, an agnostic hardware operating systems (OSes) framework for reconfigurable IoT devices. The framework provides hardware acceleration for kernel services of different IoT OSes by leveraging the RISC-V open-source instruction set architecture (ISA). The ChamelIoT hardware accelerator can be deployed in a tightly- or loosely-coupled approach and implements the following kernel services: thread management, scheduling, synchronization mechanisms, and inter-process communication (IPC). ChamelIoT allows developers to run unmodified applications of three well-established OSes, RIOT, Zephyr, and FreeRTOS. The experiments conducted on both coupling approaches consisted of microbenchmarks to measure the API latency, the Thread Metric benchmark suite to evaluated the system performance, and tests to the FPGA resource consumption. The results show that the latency can be reduced up to 92.65% and 89.14% for the tightly- and loosely-coupled approaches, respectively, the jitter removed, and the execution performance increased by 199.49% and 184.85% for both approaches.

Keywords Internet of Things · Operating systems · Hardware accelerator · Agnosticism

Extended author information available on the last page of the article

1 Introduction

The Internet of Things (IoT) has remarkably evolved during the past years, resulting in the proliferation of smart devices (*things*) through a wide variety of sectors, e.g., healthcare (Pinto et al. 2017), automotive (Cunha et al. 2022), industrial (Sanchez-Iborra and Cano 2016), and domotics (Alexandrescu et al. 2022), among others (Oliveira et al. 2020). With the growing trend of having more nodes connected to the Internet, there has been a substantial effort towards shifting heavy computational workloads from centralized facilities to decentralized networks of devices at the edge, i.e., edge computing (Perera et al. 2014; Wei et al. 2018).

Edge devices are, by nature, resource-constrained, especially when compared to their counterparts, the servers on the cloud. However, edge computing requires these devices to have better performance and real-time capabilities to gather and process data and execute their functions without needing intervention from cloud services. Despite the increasing demand for performance, these devices are still limited by their small form factor and low-power consumption requirements, pushing the limits of what is achievable with the system's current hardware (processor, memory, peripherals, among others) (Cao et al. 2020).

Aiming to improve and expand the capabilities of the current IoT edge devices, reconfigurable platforms are gaining traction in the IoT industry. These platforms, namely Field Programmable Gate Arrays (FPGAs), enable the development of custom solutions by offloading intensive software tasks to hardware accelerators (Pena et al. 2017). System configurations based on FPGAs often include a microcontroller unit (MCU) and reconfigurable fabric where the hardware accelerators are deployed. The MCU can be a hard core, i.e., implemented in silicon, or a soft core, deployed in the FPGA fabric, and it is responsible for managing the hardware accelerators. The most common targets for hardware acceleration are tasks requiring great amounts of processing power, e.g., mathematical or artificial intelligence (AI) algorithms (Boutros et al. 2020), or tasks that execute very frequently during the standard system's workflow, such as kernel services (Gomes et al. 2016; Maruyama et al. 2014). Furthermore, hardware accelerators have been used in the IoT context to optimize power consumption by applying techniques like Dynamic Voltage and Frequency Scaling (DVS) (Chéour et al. 2019; Karray et al. 2018).

Operating Systems (OSes) are widely present across embedded and IoT systems. They provide a plethora of advantages to the development of any application, for instance, managing complexity by abstracting the low-level details from the developer and providing a set of libraries that enable easy implementation of features like network communication or device drivers. Given the ubiquity of OSes in the IoT and considering that kernel services are executed countless times throughout the execution of any application, they are a prime target for hardware acceleration (Gomes et al. 2016). Nevertheless, this approach has been disregarded by the IoT industry since prior hardware accelerators were highly tailored to a specific application or OS and did not provide an easy-to-use software

application programming interface (API). An alternative solution is the implementation of a hardware accelerator capable of improving the performance of multiple OSes with only minimal modifications to their kernels and providing agnosticism to the developer by allowing applications to be executed without any changes to the code. This approach would minimize the knowledge required to build and deploy an entire system stack with hardware acceleration (Ong et al. 2013).

Another consideration regarding hardware acceleration is the processing system that manages and controls the accelerator (Maruyama et al. 2014). Depending on the target platform, this system can either be a hardcore MCU, implemented in silicon, parallel to the FPGA, or a softcore instantiated in the same reconfigurable logic as the hardware accelerator. A hardcore MCU imposes any accelerator to be connected through the resources available, which often are memory interfaces, resulting in loosely-coupled accelerators (Maruyama et al. 2010). On the other hand, a softcore MCU allows the deployment of loosely-coupled accelerators and enables the inclusion of tightly-coupled accelerators integrated directly into the MCU datapath. Additionally, if the architecture's instruction set architecture (ISA) is proprietary, the deploying method, soft or hardcore, is irrelevant. Any modification to the MCU is unfeasible due to the intellectual property being close-source, and for the same reason, scaling the accelerator to be deployed in silicon also becomes impossible.

Open-source ISAs, like RISC-V, have been rising in popularity to mitigate this effect. RISC-V is a novel open-source ISA that follows a reduced instruction set computer (RISC) design (Asanovic et al. 2014; Waterman 2016). It was designed to support a broad range of devices, spanning from high-performance application processors to low-power embedded microcontrollers. RISC-V enables a new level of software and hardware freedom by allowing easy integration of dedicated and custom-tailored accelerators with the application software. Among the multiple available implementations of the RISC-V ISA, some already take into consideration the possibility of adding accelerators as coprocessors coupled to the datapath by defining a subset of instructions for these coprocessors. With these instructions and the standard memory interfaces, implementations like the Rocket core (Asanović et al. 2016) allow for easy deployment of tightly- and loosely-coupled hardware accelerators. Consequently, RISC-V cores provide an extra degree of flexibility and the possibility of exploring the trade-offs from the two coupling approaches regarding performance, determinism, real-time, system integration, and portability (Davide Schiavone et al. 2017; Fritzmann et al. 2020; SEMICO Research Corporation 2019).

This paper presents a solution that consolidates hardware acceleration of IoT OSes in an easy-to-use agnostic framework from reconfigurable IoT devices, named ChamelIoT. Our framework leverages the Rocket core implementation of the RISC-V ISA to provide a highly configurable hardware accelerator for IoT OS kernel services, requiring minimal modifications to the OS software kernels and no modifications to application-level code. This way, there are no barriers to using ChamelIoT from the software development point of view, allowing legacy applications to take advantage of the benefits of hardware acceleration. The main contributions of this article are summarized as follows:

1. An agnostic framework that currently supports three different IoT OSes, RIOT, FreeRTOS, and Zephyr, requiring few changes to the kernel while keeping the end-user interface unmodified;
2. A highly configurable hardware accelerator with two different coupling approaches, tightly and loosely, integrated with an open-source implementation of the RISC-V ISA, Rocket core;
3. An easily portable abstraction layer for several kernel services that enable the use of hardware acceleration to any IoT OS;
4. Complete evaluation of the three aforementioned IoT OSes using the ChamelIoT framework, including microbenchmark experiments, system-wide benchmarks, and FPGA resource consumption measurements.

2 Background and related work

At the edge of the IoT ecosystem, low-end devices are becoming more strained with the increasing demands from the growing popularity of edge computing (Wei et al. 2018). Considering the nature of these devices, with limited form factor and energy consumption, frequently it is impossible to add hardware to compensate for the lack of performance. Hence, reconfigurable platforms emerge as possible solutions to IoT low-end devices (Pena et al. 2017). These platforms incorporate FPGA fabric, allowing for offloading software tasks to hardware, where they can be sped up, vastly increasing their performance and determinism. Hardware acceleration is a widely known endeavor that has proven to provide significant advantages since the early 1990s (Baum and Winget 1990; Brebner 1996). Recently, this technique has also reached the IoT across different ecosystems: security (Gomes et al. 2022; Johnson et al. 2015), AI (Qiu et al. 2016; Zhang et al. 2017), wireless connectivity (Engel and Koch 2016; Gomes et al. 2017), and image processing (DivyaKrishna et al. 2016), among others (Najafi et al. 2017; Zhao et al. 2019).

Nevertheless, reconfigurable platforms and hardware acceleration have failed to gain traction in the IoT industry until recent years since FPGA-based platforms were costly regarding price, form factor, and energy consumption. However, the latest efforts in reconfigurable fabric technology development are attempting to solve the size, weight, power, and cost (SWaP-C) constraints imposed on IoT systems. For example, Embedded FPGAs such as those provided by QuickLogic or low-power FPGAs from the Lattice portfolio are seeing increasing applicability in low-end IoT devices. These platforms are highly tailored towards low-end applications, with optimized reconfigurable slices, fewer resources available, and power-efficient organization. Further along, we expect this technology to evolve to accompany the growing need for low-end FPGAs (Allied Market Research 2023).

Traditionally, hardware accelerators were developed in FPGAs that physically were separated from the processing system, which often is an MCU where the main software application is executed. The communication methods between the two elements were often limited to the facilities available in the processing system, for instance, specialized I/O connections or memory buses. When the accelerator is connected through a standard memory interface, like AXI (ARM 2013) or TileLink

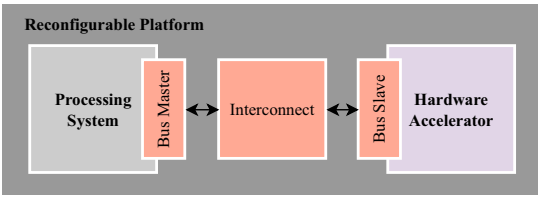


Fig. 1 Example of a loosely-coupled accelerator

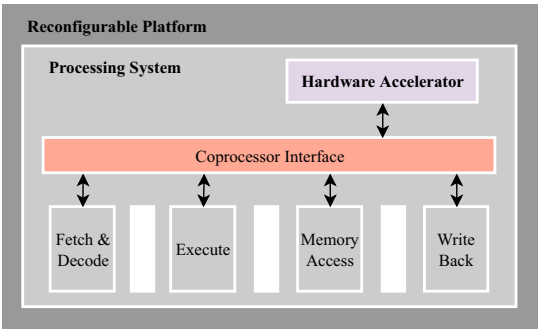


Fig. 2 Example of a tightly-coupled accelerator

(SiFive 2018), the accelerator is a memory-mapped device accessible by the software through conventional memory instructions, e.g., loads and stores. This method is referred to as a loosely-coupled approach and is depicted in Fig. 1. One of the main advantages of this approach is the easy portability across platforms since the accelerator's core can remain untouched in the porting process. Only the connection interface needs modifications to be compatible with the MCU. However, resorting to memory buses to communicate with the accelerator can lead to bus contention. Both the processing system and the hardware accelerator try to access the memory bus, consequently inducing stalls in the pipeline, which translates into worse throughput and less determinism for the whole system (Maruyama et al. 2014).

Alternatively, hardware accelerators can be implemented following a tightly-coupled approach, as illustrated in Fig. 2. This approach requires the accelerator to be integrated with the MCU datapath, forcing modifications on the pipeline, and consequently, the MCU has to be provided as a softcore. A tightly-coupled accelerator requires the MCU to include additional specialized instructions dedicated only to communicating with the accelerator. Most compilers do not accommodate these instructions by default. Hence an extra layer of abstraction is required for the software to use the instructions. On the other hand, this approach can vastly improve the system performance and determinism since the usage of external buses introduces no delays.

The two aforementioned approaches present portability, scalability, performance, real-time, and determinism trade-offs. To explore these trade-offs, it is necessary to have a reconfigurable platform that can be modified as needed. Open-source ISAs like RISC-V offer great potential since they consolidate, in a single place, the possibility of having both tightly- and loosely-coupled accelerators.

RISC-V is a highly modular and customizable open-source ISA initially developed by the University of California, Berkeley, and currently administered by the RISC-V International (Asanovic et al. 2014; Waterman 2016). It supports multiple word sizes (32 and 64-bit) and well-defined, documented, and maintained ISA extensions. These extensions can be used to tailor the ISA implementations to fit the system's requirements and constraints, leading to a wide variety of readily available RISC-V softcores, like Rocket, BOOM, CVA6, and Ariane, among others. Different implementations differ in terms of the implemented microarchitecture (e.g., pipeline stages, caches, etc.) to the high-level SoC elements (e.g., peripherals, buses).

Among these implementations, Rocket was chosen for the current stage of ChamelIoT since it provides, by default, mechanisms that enable the design and deployment of hardware accelerators, both tightly- and loosely-coupled (Sá et al. 2022). For the former approach, Rocket provides the Rocket Custom Coprocessor (RoCC) interface (Asanović et al. 2016; Pala 2017), which is integrated with the MCU datapath and accounts for up to four separate accelerators with their respective instruction opcodes. The RoCC interface also provides memory access to the accelerator without needing the MCU or software intervention. Regarding the loosely-coupled approach, Rocket allows for the inclusion of accelerators or any form of peripheral as memory-mapped devices accessible from the MCU through load and store instructions. Additionally, with the available Rocket libraries, any accelerator can also include a Direct Memory Access (DMA) port, enabling it to perform memory transactions without requiring external intervention.

Leveraging reconfigurable platforms to implement OS hardware acceleration is not a new endeavor. Moreover, the rising popularity of the IoT made it clear that it is crucial to have high-performing and deterministic OSes (Pena et al. 2017; Silva et al. 2019). There are two distinct types of OSes that use hardware acceleration: *Reconfigurable OSes* and *Hardware-accelerated OSes*. Reconfigurable OSes can be described as software OSes enhanced with capabilities to execute and schedule application-level tasks in hardware. In some cases, these OSes can also use partial reconfiguration techniques to change the accelerator in run-time according to the application's needs. On the other hand, hardware-accelerated OSes take advantage of FPGA platforms to enhance the performance of their kernel services, e.g., scheduler, thread management, or synchronization and inter-process communication (IPC) mechanisms, by migrating them to hardware.

In the following sections, we provide a brief description of several OS representatives of the two aforementioned categories. More hardware-accelerated OSes are studied and analyzed since ChamelIoT is a framework that enables hardware acceleration for kernel services. Additionally, Table 1 summarizes the gap analysis among hardware-accelerated OSes, providing a feature comparison between these OSes and ChamelIoT, highlighted in bold. For each OS, the following characteristics are identified: (1) the target CPU architecture, (2) the coupling approach, either tightly- or loosely-coupled, (3) whether the accelerator targets a Commercial Off-The-Shelf (COTS) OS or is used as a standalone OS, (4) the type of API provided, if it follows a standard like POSIX, if it can be ported to replace APIs from a software

Table 1 Gap analysis

Hardware OS	CPU architecture	Coupling approach	Supported OS	API	Scheduling	Thread management	Synchr. mechanism	IPC
HThreads	Microblaze	Loosely	Itself	POSIX	✓	✓	✓	✓
ARPA-MT	MIPS32	Tightly	Itself	Custom	✓	✓	✓	✗
HartOS	Microblaze	Loosely	Itself	Custom	✓	✓	✓	✗
SEOS	Microblaze	Loosely	Multiple	Mapped	✓	✓	✓	✓
RT-SHADOWS	Arm	Tightly	uCOS-II, FreeRTOS	Mapped	✓	✓	✓	✗
OSEK-V	RISC-V	Tightly	Itself	Custom	✓	✗	✓	✗
ARTESSO-LC	Arm	Loosely	Itself	Mapped	✓	✓	✓	✓
ARTESSO-TC	Arm	Tightly	Itself	Mapped	✓	✓	✓	✓
ChamelIoT	**RISC-V**	**Tightly and loosely**	**RIOT, FreeRTOS, Zephyr**	Mapped	✓	✓	✓	✓

OS, or if it has custom-built API, and, lastly, (5) the kernel services accelerated in hardware.

2.1 Reconfigurable operating systems

R3TOS Iturbe et al. (2015) presented R3TOS, which leverages FPGA reconfigurability to provide a reliable and fault-tolerant OS. This reconfigurable OS is composed of a multilayered architecture including a Real-time Scheduler, Network-on-chip Manager, Allocator, Dynamic Router, Placer, Diagnostic Unit, and Inter-Device Coordinator. These layers schedule the hardware tasks, manage resources, and control the access port to reconfigure the FPGA fabric. The abstraction layer provided by R3TOS aims for a "software look and feel" while alleviating the application developer from dealing with occurring faults and managing the FPGA's lifetime.

ReconOS Introduced by Lübbers and Platzner (2009), ReconOS is a Real-Time Operating System (RTOS) that allows the scheduling of hardware threads among software threads. Each hardware thread is assigned to a reconfigurable slot that encompasses two modules. The first manages the communication with software (OS interface). And the second is responsible for ensuring that hardware threads can correctly access synchronization and communication mechanisms implemented in software (OS synchronization finite state machine). ReconOS provides a POSIX-like API and a set of VHDL libraries for OS communication and memory access. Together with a system-building tool, it is possible to generate a fully integrated hardware-software project.

2.2 Hardware-accelerated operating systems

HThreads Agron et al. (2006) proposed HThreads, a multithreaded RTOS kernel for hybrid FPGA/CPU systems. This work intended to offload the (i) thread manager, (ii) scheduler, (iii) mutex manager, and (iv) interrupt scheduler to hardware. Each module is connected to the CPU through the available peripheral bus, allowing the software to access the hardware modules via load/store instructions. HThreads also allows for user-defined hardware threads that execute as a service available to software threads. Additionally, the API provided is compatible with the POSIX thread standard.

ARPA-MT ARPA-MT (Oliveira et al. 2011) is a MIPS32 implementation that takes advantage of user-defined coprocessors and exception interfaces to implement hardware support to an RTOS. The coprocessor includes a scheduler, task manager, synchronization, and communication mechanisms and provides support for non-real-time, soft, and hard real-time tasks. An object-oriented API enables the interface between software tasks and the hardware coprocessor. ARPA-MT also provides software implementations for services instantiated in hardware.

HartOS HartOS (Lange et al. 2012) is a microkernel-structured RTOS implemented in hardware that targets hard real-time applications. A custom processor, connected to the CPU through a standard peripheral bus, is responsible for

interpreting the software requests and controlling the remaining hardware modules to attend to the received requests. This custom processor shares with the remaining hardware blocks an internal memory implemented in the FPGA fabric. This memory also interacts with the timer module, watchdog module, mutexes, and semaphores, among others.

SEOS SEOS (Ong et al. 2013) is a hardware-based OS designed to provide high adaptability for easy hardware RTOS adoption. SEOS aims for easy integration with a variety of CPUs. As such, it is connected to the core through a configurable peripheral bus that meets the CPU architecture. Furthermore, SEOS provides parametrization of several modules, e.g., mutexes, semaphores, and message queues. Regarding the software, SEOS defines a set of porting steps that do not require in-depth knowledge of both SEOS and the RTOS, enabling easy integration of this hardware RTOS with a software one.

RT-SHADOWS RT-SHADOWS (Gomes et al. 2016) is an architecture that provides unified hardware-software scheduling by manipulating an ARM-based soft-core, developed in-house, to include support for multi-threading in the datapath. RT-SHADOWS is implemented as a coprocessor that includes a scheduler, thread manager, context switching, and synchronization and communication mechanisms. It leverages magic instructions (supported by unmodified compilers) to enable multiple APIs directly mapped to RTOS APIs, allowing for effortless integration in software by swapping both calls.

OSEK-V OSEK-V (Dietrich and Lohmann 2017) explores the hardware-software design space for event-triggered fixed-priority real-time systems at the hardware-OS boundary. By modifying the whole pipeline of a RISC-V core and introducing additional instructions to the ISA, OSEK-V integrates a highly-tailored hardware RTOS. Components like hardware tasks, alarms, and the scheduling policy, are implemented only to fit the application demands. The finite state machine customization happens at the compile time, ensuring the hardware-software synchronization. This approach aims at minimizing the FPGA resource consumption while maximizing the performance by designing the hardware for the application's behavior.

ARTESSO Maruyama et al. (2014) present a study comparing the same hardware-accelerated RTOS (ARTESSO) implemented in two approaches: tightly- and loosely-coupled. ARTESSO HWRTOS (Maruyama et al. 2010) was originally developed to be an integrating part of a proprietary purpose-built CPU for TCP/IP-based applications. This implementation resorted to custom ISA instructions to enable the communication between the CPU and the hardware RTOS, and included scheduling, context-switching, event/semaphore/mailbox controllers, and an interrupt controller. With the goal of making the hardware RTOS easily portable and adaptable to industrial controllers, ARTESSO can be integrated tightly-coupled to a modified Arm core or loosely-coupled to an unmodified Arm core through standard peripheral buses. The evaluation in Maruyama's study encompasses API execution times, interrupt responses, the influence of interrupts and ticks, and UDP/IP throughput. The results show that using a hardware RTOS improves the system's performance and determinism when compared to an software-only approach. Furthermore, the tightly-coupled approach results presented are at least one order of magnitude better than the loosely-coupled one. Notwithstanding, the better

performance of the tightly-coupled approach comes at the cost of portability and integrability. The tightly-coupled approach requires a modified Arm core, while the loosely-coupled is easier to integrate with any FPGA-based platform.

3 ChamelIoT overview

3.1 Motivation and goals

As supported by the extensive work in the literature presented earlier, offloading OS kernel services to hardware is not a new effort. Considering the hardware-accelerated OSes and their features, depicted in Table 1, there is a lack of direction regarding the ideal method for implementing OS hardware acceleration for IoT systems. Some OSes, like RT-SHADOWS, provide a portable API mappable into most IoT OSes allowing applications to run unmodified. Others, like HThreads or HartOS, have a dedicated API and require applications to be developed from the ground up. The multiple OSes are also deployed in different platforms, integrated with several MCUs, and have a varying range of services in hardware and configurability points. Taking this into account, both reconfigurable and hardware-accelerated OSes have yet to draw attention in the IoT industry, considering the roadblocks they present to their adoption. The main roadblocks we identify are discussed as follows:

Software interface The software API provided by each OS influences the amount knowledge regarding the whole required by the application developer. Some hardware-accelerated OSes provide their custom-built and unique API, which increases the development and learning curves since it requires developers to learn a complete set of new APIs and develop the application from scratch. Considering that time-to-market is a driving force for the IoT industry, the additional development time imposed by the learning curve of these hardware OS APIs compels the industry to opt for COTS solutions. To solve these issues, other hardware-accelerated OSes use compatibility standards like POSIX as a basis for their APIs, allowing for better portability for legacy Linux-based applications and the development of new applications due to the community's overall familiarity with POSIX-like APIs. Lastly, the remaining OSes APIs were designed to mimic software IOT OSes' APIs and replace them seamlessly. This approach allows developers to keep using the OS they are familiar with since the compiler or other external tools select which APIs to use. Additionally, this method can also enable legacy applications to use hardware acceleration since no modification is required at the application level.

Target architecture The processing system in reconfigurable platforms provides a limited number of interfaces usable by external peripherals and devices, consequently limiting the methods for integrating hardware accelerators with the core. In addition to the fact that not all MCUs are designed with hardware accelerators in mind, the number of processing systems that can be used for OS hardware acceleration is reduced. Furthermore, similarly to the software interface, developers also have a degree of familiarity with some processor families and

architectures, which results in these being preferred by the IoT industry. Considering these facts, softcore MCUs are well-suited for IoT systems with hardware acceleration since they can be modified to include and accept hardware accelerators, both tightly- and loosely-coupled. Hardware-accelerated OSes usually are developed targeting a single CPU or architecture, which limits their overall usability and portability, hindering their adoption. The problem is even worse when the accelerators are tightly-coupled to the CPU and require modifications to the datapath. This fact makes the hardware replication in mass a challenge, as it involves redesigning the silicon, which is nearly impossible with proprietary CPU architectures, often behind a paywall. To tackle this issue, some hardware OSes are exploring open ISAs, e.g., RISC-V, as their foundation, due to their availability and openness. This approach future-proofs the system and thus enhances its scalability and possible adoption.

Application suitability Due to the heterogeneity of the IoT ecosystem, the myriad of different systems presents a variety of requirements and constraints. Notably, closer to the edge, each application is progressively more constrained regarding energy consumption and form factor, as these applications are often small sensors or actuators powered by batteries. Therefore, reconfigurable platforms in the IoT edge ideally uses the smallest FPGA available. To cope with this, hardware-accelerated OSes must provide enough configurability points to fit within the hardware constraints. As such, the OS should allow the user to modify kernel parameters, e.g., number of states or priorities, or entirely remove unused features. This is only possible if the hardware-accelerated OS allows by design for such configurability, which is not always true, as a variety of accelerators in the literature are tailored to a specific application or OS.

Aiming to increase the adoption rate of hardware-accelerated OSes in the IoT market, ChamelIoT tackles the previously identified roadblocks by providing a framework for accelerating kernel services in hardware in an agnostic fashion, allowing applications from different IoT OSes to run unmodified. To do so, ChamelIoT presents the following solutions:

- Regarding the *Software Interface*, ChamelIoT offers a minimalist set of APIs that implement low-level communication with the hardware accelerator to execute well-defined kernel services. Each function can be mapped to a kernel service in software and is replaced at compile-time, providing the benefits of hardware acceleration to any IoT OS. Since only kernel internals are modified, the application-level code is kept intact. Taking this into consideration, developers can leverage ChamelIoT without the need to learn any new set of APIs or the workings of an OS, thus making it easier to use hardware acceleration.
- The *Target Architectures* of ChamelIoT are architectures deployed in reconfigurable logic. Leveraging RISC-V, an open-source ISA and its available implementations, allows ChamelIoT to explore multiple avenues of hardware acceleration, like providing the same accelerator as both tightly- and loosely-coupled. Furthermore, with RISC-V gaining popularity in the IoT industry, there is inherent scalability for hardware accelerators implemented with the same foundation as these emerging platforms.

- Lastly, ChamelIoT offers multiple configurability points to ensure *Application Suitability*. The user can opt to: (1) have the accelerator either tightly- or loosely-coupled, (2) modify multiple kernel parameters, e.g., number of threads, priorities, and states, (3) configure feature-specific implementations like the type of semaphores or if mutexes have priority inheritance, and (4) add or remove components like mutexes, semaphores or message queues as needed. All these configurations are made before the synthesis and deployment of the accelerator, enabling the user to fully customize the system and avoid unnecessary resource consumption.

Taking into consideration, the current implementation of ChamelIoT provides hardware acceleration to several OS services, which can replace with existing services of three different software IoT OSes, without requiring modifications to the application's interface. The main goals and benefits of the ChamelIoT encompass:

Real-time and determinism As one of the main requirements of low-end IoT devices, ChamelIoT must provide hard real-time guarantees and bounded worst-case execution time (WCET). Moreover, predictability shall not be affected by any configuration on the hardware accelerator, e.g., the number of priorities or the total number of mutexes, or any application-specific detail like the number of waiting threads or their priorities.

Performance It is paramount for a hardware-accelerated system to have better performance than its software-centric implementation. ChamelIoT must ensure higher performance, regardless of coupling approach or any configuration, by executing kernel services faster than their respective standard software version, independent of the IoT OS being accelerated.

Flexibility Without enough customization, it is impossible to guarantee that a hardware accelerator is not using unnecessary resources. The framework must provide several configurability points to modify ChamelIoT in a way that the accelerator behaves precisely like the target IoT OS while minimizing resource consumption.

Agnosticism In order to lessen the learning curve associated with hardware-accelerated OSes, the ChamelIoT framework must allow the user to transparently run unmodified applications of any supported IoT OS. This is achieved by only modifying the kernels' internals and leaving the user interface untouched when using the hardware accelerator.

The current stage of development of the ChamelIoT framework incorporates a hardware-accelerated OS that can be deployed both tightly- or loosely-coupled. The accelerator implements the following kernel services: scheduling, thread management, and synchronization and communication mechanisms. It is part of the already identified future work to provide a tool to ease the process of configuring and building the complete system stack. This includes configuring and synthesizing the hardware accelerator and applying the required modifications the software OS to use the accelerator. The kernel services implemented are present across the vast majority of IoT OSes, and can be implemented and deployed as hardware accelerators without changing the behavior of the software OS. On the other hand, some OS services already deployed in hardware by works

in the literature, require more intrusive implementations capable of changing the OS execution. Consequently, forcing the developer to adapt the application code. These services are not within the scope of our framework and include:

- Interrupt management—the most common approach to manage interrupts in hardware is to trap and process them in the accelerator, interrupting the processor through a single interface when needed. This approach reduces the priority spaces of interrupts and threads to a single one, i.e., the accelerator becomes responsible for managing both interrupts and threads which share the same priority rules. This approach has proven to bring several benefits to performance and memory footprint (Hofer et al. 2009; Gomes et al. 2015). Notwithstanding, it heavily modifies the OS behavior and requires the developer to understand several implementation details. Thus, the agnostic characteristics of ChamelIoT would be invalidated, as applications would need to be developed while considering the unified priority space.
- Time management—most IoT OSes rely on platform-available timers, e.g., system tick or general purpose timers, to manage time features like the tick system, delays, or periodic events. Migrating time-related operations to hardware would require replicating the timer logic in the FPGA fabric (Gomes et al. 2016; Ong et al. 2013), resulting in redundancy and waste of FPGA resources. Furthermore, it would also require redirecting the system timer interrupt to a different source, adding a priority space solely for the timer. It could lead developers to mistakenly assign priority to their interrupts, thus, breaking the system's expected behavior, which compromises ChamelIoT's agnosticism.
- Context switching—as the most architecture-dependent feature, implementing the context switching in hardware would require extensive modifications to the CPU datapath to accommodate the different register files and other data that need to be saved and loaded when a new thread is scheduled. Even though the migration of this feature to hardware has proven to bring performance and determinism benefits (Maruyama et al. 2014), a highly-tailored core limits its flexibility and adaptability. A custom-built core is not easily adapted to other platforms, consequently limiting its reusability and adoption.
- Memory management—implementing heap and thread stacks management in hardware is a great ordeal that requires a vast amount of FPGA resources. Considering the constrained nature of IoT devices, especially the FPGA-based ones, we believe this feature makes sense to be handled in software.

3.2 Architecture

In order to achieve the goals mentioned previously, the proposed ChamelIoT framework architecture is composed of three main components: (1) a tightly- or loosely-coupled Hardware Accelerator where the kernel services are implemented in hardware; (2) an Abstraction Layer that enables the interface between the software and hardware elements in the system; and (3) a Configuration and Building tool to customize the whole system stack and abstract the user from low-level implementation

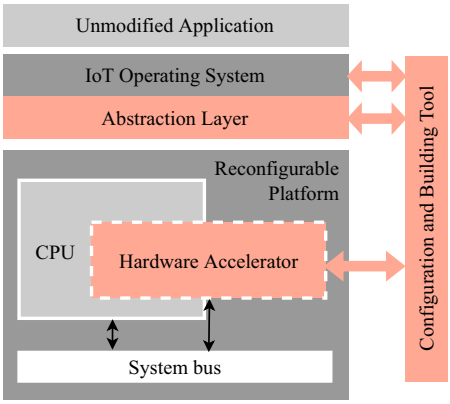


Fig. 3 ChamelIoT framework overview

details. Figure 3 depicts the three components of ChamelIoT framework incorporated in system based on a RISC-V reconfigurable platform.

Hardware accelerator This component is the central piece of the framework since it is responsible for the main goal of ChamelIoT, the acceleration of OS kernel services in hardware. The services implemented in hardware encompass (1) a scheduler responsible for determining the active thread, (2) a thread manager that stores and manages the data related to each individual thread on the system, (3) synchronization mechanisms, including mutexes and semaphores, and (4) inter-process communication mechanisms through message queues. By providing enough configurability to each service, i.e., the scheduling priority or the number of threads supported, it is possible to build the hardware accelerator to fit the application needs without wasting unnecessary FPGA resources. Designed with flexibility in mind, the Hardware Accelerator can be deployed following a loosely- and a tightly-coupled approach. The tightly-coupled approach assumes that the accelerator is integrated into the core's datapath, for instance, using a coprocessor interface. However, this option is not always available in some RISC-V implementations, where the loosely-coupled approach is always viable by connecting the accelerator to the available system bus.

Abstraction layer The software API enables the communication between the software and the hardware accelerator. This is achieved by providing a software abstraction for all the possible functions of each kernel service deployed in hardware, which results in a fine-grained abstraction layer. Together with a collection of additional APIs to access and gather context data from the accelerator, it is possible to easily adapt and port the ChamelIoT framework to most IoT OSes. Depending on the hardware accelerator coupling approach, each function in this component comprises one to four assembly instructions, a custom-made instruction for the tightly-coupled approach, and memory operations for the loosely-coupled.

Configuration and building tool Our framework intends to offer an external tool that can be used for hardware and software customization through a graphical user interface to ease the development process and soften the learning curve of hardware accelerators. This tool reduces the required knowledge about implementation details by consolidating in a single place all the customization and configuration available

in the complete system stack, automating the process of synthesizing the hardware accelerator, including the correct abstraction layer, and building the target OS with the modifications required.

4 Framework implementation

Considering the heterogeneity of applications in the IoT ecosystem, numerous IoT OSes start to implement and provide additional features to help with the variety of requirements and constraints. High-level features like wireless connectivity or cryptography are some of the prominent requirements in nowadays IoT edge devices, and OS support for these features is highly appreciated in the community. In addition to these features, design choices such as kernel architecture, scheduling policy, or synchronization and communication mechanisms have a significant influence on the developer's choice of IoT OS since these features greatly impact the overall system performance and behavior.

The OSes available for low-end IoT systems present a wide variety of them regarding implementation details and features supported. From the myriad of IoT OSes, ChamelIoT currently provides support to RIOT, Zephyr, and FreeRTOS, as they present enough variability regarding the main design points and present extensive popularity and applicability in IoT applications. The three OSes share similar design principles, e.g., they implement a preemptive priority-based scheduler and multi-queue thread management, which are also common characteristics of other low-end IoT OSes (Chandra et al. 2016; Hahm et al. 2016; Silva et al. 2019; Zikria et al. 2018). Nonetheless, there are several distinctions in their design choices which the accelerator needs to accommodate, as summarized in Table 3. The ChamelIoT framework allows for several configurations to ensure that the minimum resources are used and that the system operates exactly like the software OS. These configurations describe how the OS works, for instance, the number of thread states, the meaning of each state, the priority scheme, and which synchronization and communication mechanisms are included.

The current implementation of the ChamelIoT framework includes the Hardware Accelerator and Abstraction Layer components, while the Configuration and Building Tool are still in development. The Hardware Accelerator is based on the open-source SiFive E300, featuring an E31 Coreplex RISC-V core (RV32-IMAC), which supports atomic (A) and compressed (C) instructions for higher performance and better code density, respectively. This core is created by the Rocket Chip generator and its main characteristics include a single-issue in-order 32-bit pipeline (with a peak sustained execution rate of one instruction per clock cycle) and a single L1 instruction cache. The E300 platform also includes a platform-level interrupt controller (PLIC), a debug unit, several peripherals, and two TileLink interconnections interfaces (used to interface custom accelerators). The Abstraction Layer is composed of a set of APIs that implement low-level abstractions for the interface between the software and hardware accelerator, regardless of the coupling approach. The APIs are implemented resorting to macros and inline functions and replace the software OS APIs at compile-time.

4.1 Hardware accelerator

As the core element of the ChamelIoT framework, the Hardware Accelerator implements in hardware the kernel services common to the three IoT OSes supported, as identified in Table 2. These services include scheduling, thread management, mutexes, semaphores, and message queues, which directly correspond to hardware modules, as depicted in Figs. 4 and 5. Additionally, there is also a Control Unit module that manages the interaction between all other hardware elements and handles the interface with the processing system. Each of these hardware modules will be further detailed in later sections.

As mentioned previously, the Hardware Accelerator is implemented in a Rocket Core based platform, considering it enables the deployment of hardware accelerators both tightly- and loosely-coupled. The two different methods require different techniques and resources from the processing system, which implies modifications to hardware that manages the communication with the processing system, the

Table 2 Key features of each supported OS

OS	RIOT	Zephyr	FreeRTOS
Thread states	14	8	4
Running state	11	6	3
Ready state	12	7	2
Priority scheme	Descending	Descending	Ascending
Mutexes	Yes	Yes (with priority inheritance)	Yes (with priority inheritance)
Semaphores	Yes	Yes	Yes
Message queues	Yes	Yes	Yes
Mailboxes	Yes	Yes	No

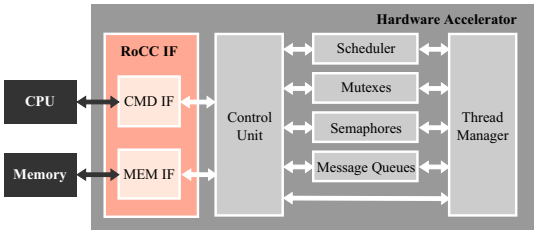


Fig. 4 Tightly-coupled hardware accelerator architecture

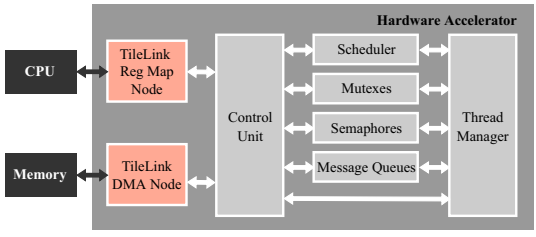


Fig. 5 Loosely-coupled hardware accelerator architecture

Control Unit. Since this module is solely responsible for the interface with the CPU, all the other hardware elements remain untouched, independently from the coupling approach.

Tightly-coupled For this approach, the Hardware Accelerator leverages the RoCC interface provided by the Rocket core to integrate the coprocessor directly into the datapath. This interface is composed of two smaller interfaces, as illustrated in Fig. 4: (1) the command interface (CMD IF), which is responsible for receiving and answering any requests from the pipeline, and (2) the memory interface (MEM IF), through which any memory transaction can be made without CPU intervention.

Along with the ISA specification, the command interface imposes restrictions on the hardware accelerator. The instruction type specified for RoCC instructions is R-type, which limits the data input to the accelerator to two 32-bit words and the output to one 32-bit word, all in general-purpose registers. This data is provided directly to the command interface already decoded by previous pipeline stages, along with a 7-bit field that works as an internal *opcode* for the accelerator. Since the accelerator is integrated within the pipeline, it also needs to follow its timing restrictions, implying that the command interface needs to have the output ready in the same clock cycle. This fact forces the output logic to be fully combinational, demanding the majority of the other modules to be implemented with combinational logic.

Loosely-coupled In this approach, the Hardware Accelerator acts as a memory-mapped device for the processing system. The accelerator needs to be registered as two different TileLink nodes, as depicted in Fig. 5: (1) a Register Map node, through which the software can write input data and read the output from predefined memory addresses, and (2) a Direct Memory Access (DMA) node used to provide memory access to the accelerator without CPU intervention.

In the TileLink Register Map node, a memory address range has to be assigned to the accelerator, which determines the addresses the software can use to communicate with the hardware. In this range, a register file is defined according to the inputs and outputs required for each kernel service. Each register is composed of a 32-bit word and has a specific address. The software application can access these registers through load and store instructions. Additionally, the TileLink DMA node grants the accelerator access to the system memory through burst operations. These operations are done in bursts with predefined sizes (values are always in powers of 2), which limits finer-grained memory transactions.

4.1.1 Control Unit

The Control Unit is the main component of the Hardware Accelerator, as it ensures that all the other elements function as intended. Among its responsibilities, the Control Unit manages the interfaces with the processing system available in the accelerator. It involves processing the software request by interpreting internal instruction opcode, collecting the input data from the correct sources, commanding other modules to execute the proper functions, and outputting results in a timely fashion. When the accelerator is tightly-coupled, both the input and output data is available through

the RoCC command interface. The software issues a single instruction containing the source and destination registers for the data. Contrarily, in the loosely-coupled approach, the input data is provided by software by storing data in the register map before the execution of any function. Likewise, the output is available in the register map to be read by the software after executing the main instruction.

Considering the limited nature of FPGA resources, there is a limit to the number of kernel services that can be deployed in hardware. Mutexes, semaphores, and message queues are kernel elements of which the software can use multiple instances. Therefore, the maximum number of these modules deployed in hardware is a parameter configured by the user and the Control Unit is responsible for managing which modules are free or used in run time.

Lastly, regardless of the coupling approach, there is only a single interface to perform memory operations. Consequently, whenever another module, e.g., message queue, needs to execute any memory transaction, it must request the Control Unit to execute that operation. This way, it is impossible to have multiple modules concurrently trying to access the system's memory. Additionally, the Control Unit has an internal memory buffer used by the message queues to store data that has not yet been requested by the software. Given that some message queue operations require direct transfer to and from this buffer to the system memory, it also becomes part of the Control Unit's responsibility to manage the buffer to avoid concurrency in any access.

4.1.2 Thread Manager

The Thread Manager is mainly responsible for storing and managing the data of each thread used by the software application. With the goal of minimizing FPGA resource usage, the total amount of active threads allowed in the system is a configurable parameter at compile time. This configuration is one of the most impacting on resource usage because it forces the internal Thread Identifier (TID) to have a field size capable of holding the highest number of threads. Taking into consideration that the TID is a field propagated throughout the whole accelerator, it naturally increases resource consumption, especially considering that most of the accelerator is implemented with combinational logic.

The TID is used by the Thread Manager to address each thread added by the software application. It represents an index in an array of *Thread Nodes*, which are structures that contain thread data required by most hardware modules, as depicted in Fig. 6. The *data* field stores a pointer to the Thread Control Block (TCB) provided by the software OS. The accelerator uses this pointer to enable multiple ways for the software to access a thread, either through TID or TCB. Furthermore, there is a context in the TCB that has not been migrated to hardware, e.g., memory management details, allowing the software OS to not be

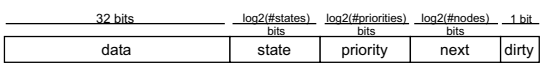


Fig. 6 Thread Node structure

limited by what ChamelIoT implements. Both *state* and *priority* fields are used by the scheduling algorithm and have variable field sizes according to additional configurability points. A single-bit field, *dirty*, is used to indicate whether or not that index is available. When a thread is added to the system, the software should provide the previous fields, and the hardware determines which index is free, then sets the *dirty* bit and outputs its index. To remove a thread, the Thread Manager only needs to clear the *dirty* bit.

The last field in the *Thread Node* is named *next*, and it is used to implement linked lists utilized in the queue of threads ready to be scheduled, henceforth referred to as ready queue. Given that all thread data is stored and handled by the Thread Manager, it is also part of this component's function to manage the state of each thread and the ready queue. The ready queue implemented in hardware follows a multi-queue system, where there is a circular linked list for each priority level, leveraging the *next* field in each node to point to the next thread with execution rights with the same priority. Figure 7 illustrates a state example of a ready queue with five threads. Thread 3 is currently running, and Thread 5 is blocked, both with the same priority level, resulting in the Thread 3 node pointing to itself. The remaining three threads have lower priority and form a circular list while waiting to be scheduled. Lastly, any changes to the ready read are prompted by changes in the thread state, which can be caused by a mutex, semaphore, or message queue blocking or unblocking a thread, the scheduling algorithm, or directly by a software request.

4.1.3 Scheduler

The scheduling policy implemented follows a preemptive priority-based algorithm that uses a hardware configuration to define the priority order, i.e., ascending or descending. By definition, the thread with the highest priority in a ready state will run until it yields its execution time or it is interrupted by a thread with higher priority. In case of multiple threads with the same priority, the scheduling algorithm follows a round-robin scheme to determine which thread is to be executed next.

To schedule the next thread, the Scheduler accesses the Thread Manager's ready queue to identify which is the highest priority among threads in a ready state. Then, the Scheduler is responsible for changing the currently active thread state from *running* to *ready* and the other way around for the new thread. Due to timing constraints, it is mandatory for the Scheduler to be implemented with only combinational logic since whenever a scheduling operation is requested, the kernel is in the process of swapping active threads, which must be deterministic and requires the shortest possible time.

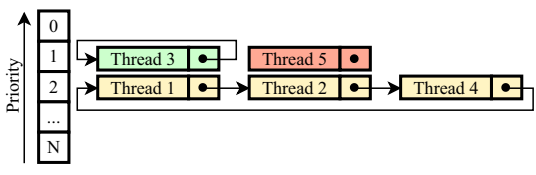


Fig. 7 Ready queue state example

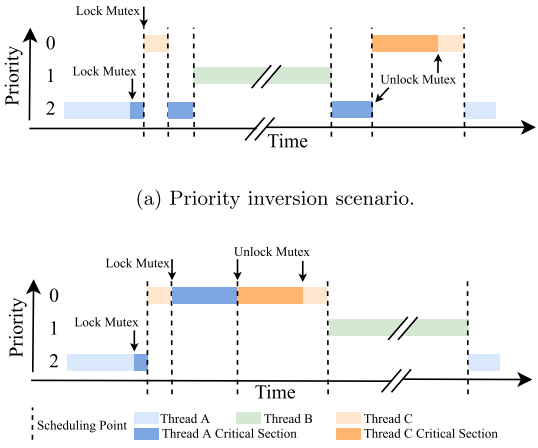
4.1.4 Mutexes

A mutex is a synchronization primitive that ensures mutually exclusive access to a resource. In the context of operating systems, a thread can use a mutex to guarantee that its access to a shared resource is undisturbed and that other threads cannot corrupt the resource. To perform an access, a thread must attempt to *lock* the mutex, which only is successful if no other thread is locking it. Once the thread is successful, it becomes the owner of said mutex and holds its ownership until the thread *unlocks* the mutex. On the other hand, if an attempt to *lock* a mutex fails, the thread that tried is blocked and yields its execution to the next thread.

In the most common implementations of mutexes in IoT OSes, a failed *lock* can result in a priority inversion scenario, where a thread with lower priority executes before one with higher priority. It can lead to an instance where a critical portion of code protected by a mutex is delayed to a later scheduling point. This is depicted in Fig. 8a, where the highest priority Thread C interrupts Thread A, and fails to lock a mutex currently owned by Thread A. This lead to a case where Thread A critical section only occurs after Thread B finishes executing. These situations are not desirable in IoT edge devices where real-time and determinism are paramount.

A possible solution to priority inversion scenarios is using priority inheritance algorithms, which consist of raising the priority level of the current owner of a mutex to the highest priority of the threads that attempted to lock the same mutex. This is represented in Fig. 8b, where after Thread C fails to lock the mutex, Thread A is given the same priority level to finish executing its critical section and *unlock* the mutex. Once Thread A *unlocks* the mutex, Thread C can resume its critical section, and Thread B only runs after the highest priority thread finishes its execution.

In ChamelIoT's hardware accelerator, each Mutex implementation maintains a register with the current thread that owns the mutex and a list of TIDs of each thread that was blocked trying to lock it. This list of threads also contains their respective priority, to enable the implementation of priority inheritance mechanisms. Whenever



(a) Priority inversion scenario.

(b) Priority inheritance scenario.

Fig. 8 Mutexes use case scenarios

a thread's priority is modified, the Mutex informs the Thread Manager of the TID and new priority. Consequently, the Thread Manager can keep the ready queue correct, removing the thread from one linked list and adding it to the list regarding the new priority. The process is the same whether the priority is raised as a result of a failed *lock* or lowered after an *unlock*. Finally, whenever a thread is forced to change state, e.g., to ready state when the priority is raised, the Scheduler is also updated accordingly, and the software is notified in the next scheduling point.

4.1.5 Semaphores

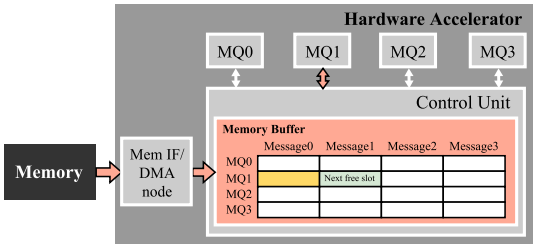
A semaphore is another method of synchronization utilized in operating systems. It follows a *producer–consumer* scheme, where the *producer* thread signals the semaphore once it finishes acquiring or processing a certain resource. Internally, the semaphore registers the count of how many signals were emitted by the *producer* thread. In turn, the *consumer* thread checks, through the semaphore, if there are resources available. If the semaphore's internal count is greater than zero, the *consumer* thread is allowed to keep executing, otherwise the thread is blocked. Semaphores are often used in cases where a resource is produced at a high frequency, e.g., data acquired from a sensor, and multiple threads need access to it.

Each hardware Semaphore has a configurable maximum count of resources produced and variable size of threads that are blocked when there are no resources available. When a thread tries to *take* from a semaphore and fails, it is blocked and its TID and priority are saved internally in the semaphore. At the same time, the Thread Manager is informed to remove the thread from the ready queue. These two values are used later when a *producer* thread issues a *give* operation to request the Thread Manager to put the blocked thread in the ready queue.

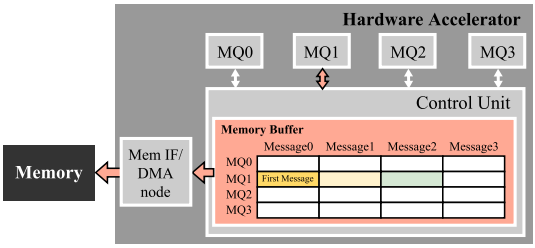
4.1.6 Message queues

Message queues are asynchronous communication mechanisms used to send data from one thread to another. Common implementations of message queues in OSes use Firs-In First-Out (FIFO) queues to store messages waiting for a receiving thread. When a thread attempts to send a message, it should provide a pointer to the data and the message size so that the message queue can store a copy of the message. Likewise, when a thread receives a message, it should provide a pointer to the address where it wants the data to be stored so it can receive a copy of the data. When the message queue holds a copy of the data, it avoids having the threads share memory, which often requires extra care to prevent data corruption.

As mentioned previously, the Control Unit manages the internal memory buffer for all the hardware Message Queues to prevent concurrent accesses to the memory. This buffer is composed of a configurable limit of messages per message queue, with the message size also being configurable. The scenarios illustrated in Fig. 9 show an example of this memory buffer when there are four message queues in the system with a limit of four messages each.



(a) Data flow in a *put* operation.



(b) Data flow in a *get* operation.

Fig. 9 Examples of scenarios with message queues operations

Figure 9a depicts a *put* operation on Message Queue 1. In this case, there is a message already stored in the buffer. The Control Unit reads the data from the system memory and stores it in the next free message slot. On the other hand, in the *get* operation, illustrated in Fig. 9b, the data written to the system memory is from the first message stored in the message queue, forming a FIFO queue. As such, the Message Queues need to keep track of the order in which the messages are stored.

Lastly, whenever a thread tries to receive a message and the buffer is empty, the thread is blocked until a message is sent. At the same time, when a thread attempts to send a message, and there is no thread waiting to receive it, i.e., if the buffer is full, the sending thread is also blocked to prevent overwriting other data. The amount of threads in each waiting list, sending and receiving, is also a configurable parameter.

4.2 Software abstraction layer

In the ChamelIoT framework, the Software Abstraction Layer fulfills the role of mediator between the software kernel and the Hardware Accelerator. This layer is mainly responsible for: (1) providing low-level generic functions that interact with the accelerator, regardless of coupling approach, (2) implementing functions for each kernel service in hardware and accessing their context data, and (3) doing the modifications needed so that the supported OSes use ChamelIoT's API.

Regarding the low-level functions that interact with the Hardware Accelerator, these have to take into consideration the coupling approach. For the tightly-coupled accelerator, all the interaction must be made in a single custom instruction, while accessing the loosely-coupled accelerator is done by reading and writing to specific addresses.

Providing support for additional operating systems in ChamelIoT requires the developer to have intimate knowledge of both the ChamelIoT framework and the

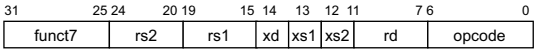


Fig. 10 RoCC instruction format

OS. To add a hardware service to an IoT OS, the developer should follow the guidelines below:

1. Identify the code blocks or functions within the kernel that implement the service;
2. Add conditional compiling verification macros;
3. Include the necessary calls to ChamelIoT API to replicate the service behavior;
4. Ensure the inputs and outputs of the ChamelIoT Abstraction Layer match the software version.

Furthermore, additional modifications may be required to the Software Abstraction Layer. This is particularly applicable in unique scenarios where the operating system requires specific inputs, outputs, or functionalities that are not readily available in the hardware or require significant alterations. In this case, the suggested approach is to keep the additional features outside the API to ensure the behavior remains unchanged.

Tightly-coupled Considering that ChamelIoT's Hardware Accelerator is currently deployed in a Rocket-based platform, it leverages the RoCC interface to implement a tightly-coupled approach. Regarding the communication with the CPU, the RoCC interface defines an extension to the RISC-V ISA by introducing a custom instruction that follows the R-type format, depicted in Fig. 10. It specifies the target coprocessor, the source and destination of data, and the performing operation.

The *opcode* field identifies the coprocessor, and according the RoCC specification, it can only contain one of four predefined values, thus, limiting the number of coprocessors. The fields *rd*, *rs1*, and *rs2* specify the destination (rd) and source (rs1 and rs2) CPU registers used to transfer data with the coprocessor. The *xd*, *xs1*, and *xs2* are auxiliary fields that identify which of the previous registers have valid data. Lastly, the field *funct7* is used as a user-defined opcode for each coprocessor that indicates which function has to be executed.

Listing 1 demonstrates how the RoCC instruction is translated into a C macro. This macro is then used in the implementation of kernel service APIs by having function arguments directly mapped to the source registers *rs1* and *rs2*, and the return value coming from the *rd* register. The *funct7* is determined by a table which maps every function implemented in hardware to a unique value. In order to keep every service available through a single instruction, the interaction between the CPU and accelerator becomes limited to: (1) two 32-bit words being received on the coprocessor; (2) a single 32-bit word response; and (3) a maximum of 128 distinct operations.

```
#define ChamelIoT_opcode 0b1011011
#define ROCC_INSTRUCTION(rd , rs1 , rs2 , func7) \
    __asm__ volatile (".insn r " STR(ChamelIoT_opcode) ", \
    " STR(0x7) ", " STR(funct7) ", %0, %1, %2" \
        : "=r"(rd) \
        : "r"(rs1), "r"(rs2))
```

Listing 1 C macro for the RoCC instruction

Loosely-coupled When the accelerator is deployed loosely-coupled, it is integrated with the Rocket core as a TileLink Register Map node. Consequently, the Hardware Accelerator becomes a memory-mapped device with a well-defined range of addresses configured at compile-time. This implies that the accesses to the accelerator from the software application are done via loads and store instructions. An example of code used to execute these instructions is depicted in Listing 2, where a read and write to specific accelerator registers.

The register map consists of a register per possible input and output and a special register for the instruction. In the current version of the hardware accelerator, the register map includes a total of 34 registers. The instruction register is located at the accelerator base address, and it triggers the accelerator to perform a function whenever anything is written in this register. This approach uses the same *funct7* values to offer a similar behavior to the RoCC interface implementation, allowing most of the hardware component to remain unmodified.

The interaction between software and hardware is done through memory accesses. It provides flexibility regarding inputs and outputs as they are not limited by the boundaries of a single instruction. However, it comes at the cost of needing more instructions to execute a single service which involves writing all the inputs, then writing the instruction register, and finally reading the outputs.

Independently from the coupling approach, the Software Abstraction Layer provides a library of functions to be used by IoT OSes in their kernels. Table 3 lists all the current APIs used to accelerate the current IoT OSes. These include functions to add or remove threads from the system, change thread states, schedule a new thread, and use any operation in mutexes, semaphores, and message queues. Furthermore, some functions allow the software to collect data from any hardware component in the accelerator.

As mentioned previously, currently, the ChamelIoT framework supports three IoT OSes: RIOT, Zephyr, and FreeRTOS. For each OS, minimal modifications had to be made in their kernels to use ChamelIoT's accelerator. Listing 3 shows an example of a modification made to Zephyr's kernel, replacing the code to schedule the next thread. Currently, this is achieved by resorting to preprocessor directives, allowing the user to decide if hardware acceleration is used in the system by defining a variable during the OS building process. Furthermore, any additional feature not supported by the API that needs to be included to the system should be added within the preprocessor directives.

```
1 #define CHAMELIOT_I_TID 0x1001A00C
2 #define CHAMELIOT_O_TID 0x1001A048
3
4 *(unsigned int *)(CHAMELIOT_I_TID) = data;
5 value = *(unsigned int *)(CHAMELIOT_O_TID)
```

Listing 2 Write and read data to the loosely-coupled accelerator

Table 3 List of functions available is the software abstraction layer

Function	Inputs	Outputs
Add thread	Priority, TCB	TID
Remove thread	TCB	–
Set thread state	State, TCB	–
Schedule	–	TID
Get active thread TCB	–	TCB
Get active thread TID	–	TID
Convert TCB from TID	TID	TCB
Conver TID from TCB	TCB	TID
Initialize mutex	Mutex ID	Success or error
Lock mutex	Mutex ID	Success, error or schedule
Unlock mutex	Mutex ID	Success, error or schedule
Initialize semaphore	Semaphore ID	Success or error
Give semaphore	Semaphore ID	Success, error or schedule
Take semaphore	Semaphore ID	Success, error or schedule
Initialize message queue	MQ ID	Success or error
Put message queue	MQ ID, memory address	Success, error or schedule
Get message queue	MQ ID, memory address	Success, error or schedule

```
#ifdef CHAMELIOT
  struct k_thread *thread = (struct k_thread *)
    ChamelIoT_get_TCB(ChamelIoT_schedule());
#else
  struct k_thread *thread = next_up();
#endif //CHAMELIOT
```

Listing 3 Example of kernel modifications to use ChamelIoT's API

5 Evaluation

For the purpose of evaluating the ChamelIoT framework, we have integrated and provided support for three IoT OSes: RIOT, FreeRTOS, and Zephyr. To assess performance and determinism, we measured the latency of most kernel services APIs through a series of microbenchmark experiments that measured the clock cycles required by most kernel services implemented by the hardware accelerator. We also evaluated the overall system's performance using the Thread Metric benchmark suite, which provides a set of synthetic benchmarks stressing kernel features, like scheduling, and synchronization. Each experiment was performed for the three OSes targeting the multiple configurations available with the ChamelIoT framework:

- SW—the software-based approach without using the hardware accelerator available, i.e., the vanilla software implementation of each OS;
- TC—the tightly-coupled approach where the hardware accelerator is connected to the core through the RoCC interface, and each OS uses the hardware acceleration by using specific instructions;
- LC—the loosely-coupled approach, where the multiple OSes leverage the memory-mapped hardware accelerator through memory operations.

Additionally, we evaluated the impact of ChamelIoT on the hardware resources and power estimation required by different threads and priorities configurations, which (from empirical experiments) are the most impactful configurability points. Lastly, we measured the memory footprint of each OS for the three ChamelIoT configurations.

5.1 Experimental setup

We deployed and evaluated our solution on an Arty A7-100T, which features a Xilinx XC7A100TCSG324-1 FPGA running at a clock speed of 65MHz. The hardware accelerator is integrated into an E31 Coreplex RISC-V core (RV32-IMAC). Both the RISC-V core and our accelerator were implemented using the SiFive Freedom E300 Arty FPGA Dev Kit and synthesized in Vivado 2020.2.

The performance evaluation experiments targeted the RIOT v6ae67, FreeRTOS v10.2.1, and Zephyr v2.6.0-577. All software was compiled with the GNU RISC-V Toolchain (version 9.2.0), with optimizations for size enabled (-Os). Apart from OS-specific configurations such as the priority order, the hardware accelerator was kept with the same configurations for the three OSes: maximum of 16 threads with 16 unique priorities, four different mutexes, semaphores, and messages queues (each with 16-word size, and a list of four messages).

5.2 API latency

To assess determinism and performance, we have measured the number of clock cycles required to execute the most common OS services for the three aforementioned setups. Each experiment was repeated 10,000 times for each kernel service. The results are presented by the average number of cycles, i.e., arithmetic mean (M), along with the standard deviation (SD) measured across all repetitions. Furthermore, the worst-case execution time (WCET) measured across all the experiments is presented for each API. The results are discussed below.

Table 4 Thread Manager and Scheduler API latency

		Thread suspend		Thread resume		Schedule	
		M ± SD	WCET	M ± SD	WCET	M ± SD	WCET
RIOT	SW	57.06 ± 0.68	66	59.08 ± 1.03	75	78.99 ± 0.37	82
	TC	32.00 ± 0.00	32	35.01 ± 0.19	38	12.00 ± 0.00	12
	LC	57.06 ± 0.57	63	123.96 ± 0.50	127	30.00 ± 0.00	30
Zephyr	SW	132.98 ± 0.64	138	110.00 ± 0.00	110	39.98 ± 1.22	48
	TC	115.02 ± 0.27	118	108.00 ± 0.00	108	20.01 ± 0.19	23
	LC	212.04 ± 0.63	226	293.02 ± 0.38	299	32.01 ± 0.19	35
FreeRTOS	SW	221.67 ± 3.09	240	107.06 ± 0.65	113	462.35 ± 1.68	464
	TC	79.14 ± 0.87	85	42.02 ± 0.38	48	55.02 ± 0.27	58
	LC	107.06 ± 0.42	110	60.03 ± 0.44	67	69.00 ± 0.00	69

5.2.1 Thread Manager and Scheduler

Table 4 presents the results regarding the latency of three different APIs implemented by the Thread Manager and Scheduler: *Thread Suspend*, *Thread Resume*, and *Schedule*. The first two APIs are responsible for modifying the thread state, i.e., from ready to suspended state in *Thread Suspend* and the other way around in *Thread Resume*. Whenever one of these functions is executed by a kernel, it implies adding or removing a thread from the ready queue. To test these functions, the system included two threads with different priorities, where the higher priority thread suspends itself, and the low-priority thread resumes the high-priority thread. Lastly, the *Schedule* function is executed at every scheduling point to select the next executing thread. In order to test this function, two threads with the same priority constantly yielded their execution time to trigger an explicit scheduling point.

The TC setup on RIOT presents latency decreases and improved determinism on all three APIs when compared to the baseline (SW configuration). The latency is decreased by 43.92% on *Thread Suspend*, 40.73% on *Thread Resume*, and 62.02% on *Schedule*. Additionally, the standard deviation is closer to zero on all kernel service, indicating better determinism. On the other hand, the LC setup only shows better performance on the *Schedule*, decreasing its latency by 62.02%. Nonetheless, it presents better determinism. The number of cycles required to perform a *Thread Resume* on the LC configuration is over double the number required on the SW configuration. This is justified by the fact that the Abstraction Layer for the loosely-coupled accelerator requires multiple registers to be written in order to perform a service or access a value from the accelerator, e.g., thread priority or state. If a function performs multiple accesses like these, it will greatly increase the total number of cycles required by that API since both the SW and TC would only need a single instruction to execute the same function.

The results gathered for Zephyr also show that only the TC setup increases the performance over the SW setup. The latency is decreased by 13.51%, 1.81%, and 49.94% *Thread Suspend*, *Thread Resume*, and *Schedule*, respectively. Furthermore, the standard deviation is also lower. The LC configuration also shows a performance increase of 99.76% on the *Schedule* function and better determinism on all three APIS. However, for the previously mentioned reasons, both *Thread Resume* and *Thread Suspend* functions show latency increases.

Lastly, both hardware-accelerated setups on FreeRTOS present performance increase on all three kernel services along with less variance. On the TC setup, the latency is decreased by 64.30% on *Thread Suspend*, 60.75% on *Thread Resume*, and 88.10% on *Schedule*. And the LC setup decreases the latency by 51.70%, 43.93%, and 85.08% for each API, respectively.

5.2.2 Mutexes

Table 5 summarizes the results gathered for the two APIs related to Mutexes in three different scenarios. The *Lock* function is used when a thread attempts to acquire the mutex before entering a critical section of code. Whenever this function executes three different scenarios can occur: (i) the mutex is successfully locked, and the

Table 5 Mutexes API latency

		Mutex lock						Mutex unlock					
		Success		Priority inherit.		Fail		Success		Priority inherit.		Fail	
		M ± SD	WCET	M ± SD	WCET	M ± SD	WCET	M ± SD	WCET	M ± SD	WCET	M ± SD	WCET
RIOT	SW	22.00 ± 0.00	22	–	–	65.14 ± 3.25	72	23.96 ± 0.33	24	–	–	17.04 ± 0.48	24
	TC	19.00 ± 0.00	19	–	–	21.00 ± 0.00	21	18.02 ± 0.27	21	–	–	20.95 ± 0.38	21
	LC	49.00 ± 00	49	–	–	53.00 ± 00	53	49.99 ± 0.19	50	–	–	50.00 ± 0.00	50
Zephyr	SW	30.02 ± 0.27	33	70.03 ± 0.37	75	33.03 ± 0.21	35	78.04 ± 0.63	88	88.96 ± 0.58	89	12.01 ± 0.19	15
	TC	23.00 ± 0.00	23	23.00 ± 0.00	23	24.00 ± 0.00	24	12.02 ± 0.27	15	12.01 ± 0.19	15	12.00 ± 0.00	00
	LC	55.00 ± 0.00	55	45.01 ± 0.84	49	59.00 ± 0.00	59	45.01 ± 0.84	49	45.65 ± 1.06	50	45.07 ± 0.48	49
FreeR-TOS	SW	100.12 ± 1.29	115	939.07 ± 1.78	955	107.23 ± 1.02	126	123.09 ± 0.98	137	303.51 ± 1.56	309	75.20 ± 1.28	88
	TC	66.06 ± 0.62	73	69.03 ± 0.45	76	68.00 ± 0.00	68	66.02 ± 0.27	69	69.98 ± 1.12	75	68.92 ± 0.68	75
	LC	91.00 ± 0.00	91	102.03 ± 0.27	105	95.00 ± 0.00	95	90.02 ± 0.27	93	107.03 ± 0.29	110	90.02 ± 0.27	93

current thread continues to execute; (ii) the mutex is already locked, and the priority inheritance mechanism is triggered; and (iii) the API fails to lock the mutex for external reasons, e.g., uninitialized or invalid mutex. The *Unlock* API is used when a thread is leaving a critical code section to release the mutex ownership. Likewise, this service can result in three different scenarios: (i) a successful unlock, where the thread keeps execution rights; (ii) the thread had its priority raised by the priority inheritance mechanism, and consequently, its priority has to be reverted, and a scheduling point is forced; and (iii) the *Unlock* fails because the mutex was not previously locked, for instance. In order to test the three scenarios, we first had a single thread successfully locking and unlocking the same mutex and then trying to lock and unlock an uninitialized mutex, leading to failed attempts on both operations. Lastly, to validate the priority inheritance scenario the following steps are executed in a loop: (1) a low-priority thread locks a mutex and resumes a high-priority thread; (2) the high-priority thread attempts to lock the mutex, triggering the priority inheritance mechanism, forcing the other thread to execute; (3) the first thread has its priority raised and unlocks the mutex, once again triggering the priority inheritance to revert its priority and schedule the next thread; and (4) the high priority thread unlocks the mutex and suspends itself. The results shown in Table 5 for this scenario were collected in the *Lock* function in step 2 and the *Unlock* in step 3.

RIOT does not support priority inheritance in its mutex implementation, as such, no results are available for this scenario. The TC configuration shows latency decreases of 13.58% on successful *Locks*, 70.81% on failed *Locks*, and 24.79% on successful *Unlocks*. On failed *Unlocks*, this setup shows a minimal latency increase. However, it presents lower standard deviation and a better WCET. The LC setup presents performance degradation on most scenarios due to the increasing number of instructions to communicate with the accelerator.

The TC setup on Zephyr decreases the latency on both APIs and all scenarios. For the *Lock* operations, it shows decreases of 23.35%, 67.14%, and 67.16% for the successful, priority inheritance, and fail scenarios, respectively. On the *Unlock* function, the latency is decreased by 84.59%, 86.50%, and 0.10% on the same scenarios correspondingly. For the LC setup, there is a latency increase in the successful and fail experiments on both APIs. However, on the priority inheritance scenario, it shows a decrease of 21.46% and 48.68% on the latency of the *Lock* and *Unlock* functions.

Finally, both hardware configurations, TC and LC, increase the performance on all scenarios and APIs. This is most notable in the priority inheritance cases, where for the *Lock* function the latency is decreased by 92.65% on the TC setup and 89.14% on the LC one. And for the *Unlock* function, it is decreased by 76.94% and 64.75% on the TC and LC configurations, respectively. FreeRTOS software kernel implements its ready queues in a fashion that requires the iterative traversing of linked lists whenever a TCB is accessed. This results in longer times on functions that need to modify thread priorities and states multiple times, like the priority inheritance algorithm on mutexes.

5.2.3 Semaphores

The Semaphores API consists of mainly two functions: *Give* and *Take*. The *Give* function is used by a thread to signal a semaphore that new data or resources are now available. Whenever this function executes, it can result in two different scenarios. The first where another thread previously tried to take from the semaphore, and the current thread has to yield the execution after the *Give*. And the second scenario where there is no thread waiting, and the current thread continues execution. The *Take* function is used by a thread attempting to access a resource, and similarly to the previous function, it can also result in two different scenarios: (i) the resource is already available in the semaphore, allowing the current thread to access it and keep executing; and (ii) there are no resources available, forcing the current thread to yield until the semaphore is signaled by other threads. To test these APIs, we devised two experiments, one with a single thread using *Give* and *Take* repeatedly, leading to no yields being required. And another experiment with two threads, where the first attempts to take from a semaphore without resources, yielding the execution to the other thread, which signals the semaphore and yields the execution to the waiting thread. The results collected are presented in Table 6.

For RIOT OS, the TC setup provides a latency decrease for all the scenarios in both APIs. This decrease varies from 69.16% on a *Take* with no resource available to 72.70% on a *Give* with a thread waiting. The LC setup slightly increases the latency in all cases, up to 7.84%. However, it decreases the latency variance.

The TC configuration on Zephyr decreases the latency of *Gives* with threads waiting by 63.65% and without threads waiting by 89.05%. For the *Take* API, this setup only performs slightly better (up to 4.55%) than the SW configuration. The LC setup decreases the latency on *Gives* with threads waiting by 0.12% and 75.23% on *Gives* with no thread waiting. On the *Take* API, the latency increased to over double. The software implementation of semaphores on Zephyr is already fast, to the point that hardware acceleration mostly offers better determinism.

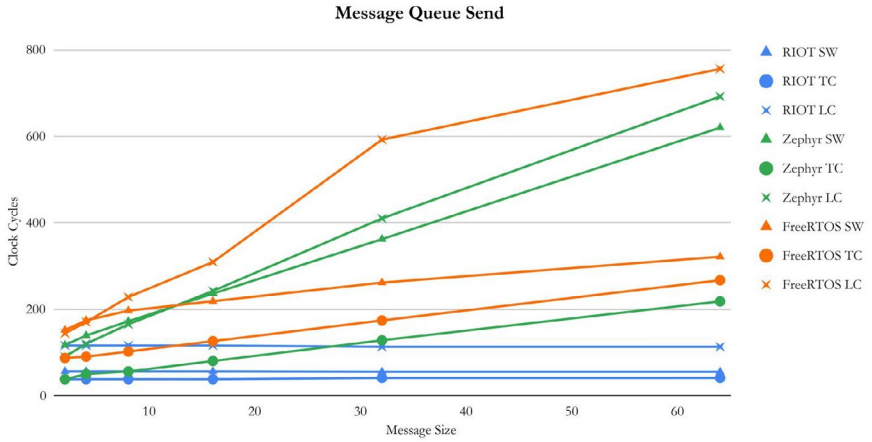
Lastly, the hardware-based setups increase the performance on all cases for the semaphores API in FreeRTOS. This is most evident in the scenarios that require the thread to yield execution, e.g., *Gives* with threads waiting (latency decreased by 54.70% and 49.67% on the TC and LC setups respectively), and *Takes* with no resource available (latency decreased by 89.58% and 87.07% on the TC and LC setups, respectively). As mentioned previously, this is due to the fact that FreeRTOS uses more complex logic to access the ready queue.

5.2.4 Message queues

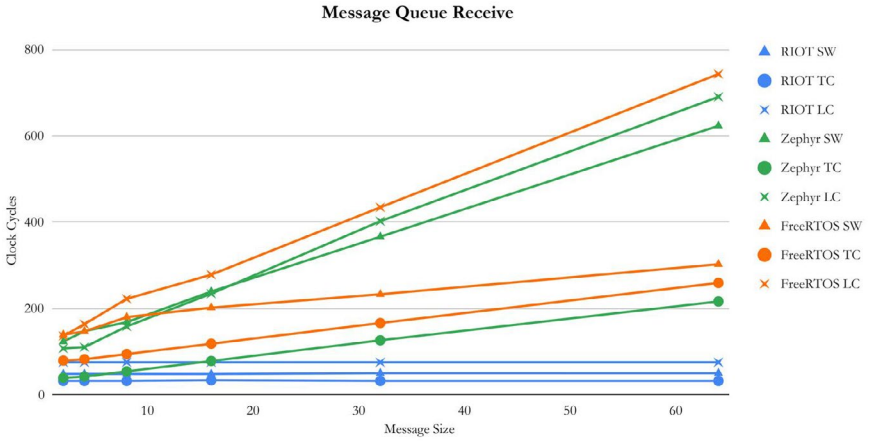
There are two main operations regarding message queues, i.e., *Send* and *Receive*. Both functions may cause threads to be suspended or resumed, depending on whether there is someone waiting for the message or if there is a message ready. However, for the purpose of isolating the memory operations and evaluating their performance, scenarios, where threads had to yield execution, were not considered. As such, the conducted experiment consisted of one thread sending a message through a message queue, and another thread receiving it through the same queue.

Table 6 Semaphores API latency

		Semaphore give				Semaphore take			
		Thread waiting		No thread waiting		Resource available		No resource available	
		M ± SD	WCET	M ± SD	WCET	M ± SD	WCET	M ± SD	WCET
RIOT	SW	44.06 ± 0.68	54	45.05 ± 0.76	57	39.02 ± 0.38	42	39.02 ± 0.38	42
	TC	12.03 ± 0.44	19	12.03 ± 0.44	19	13.00 ± 0.00	13	12.04 ± 0.33	15
	LC	44.98 ± 0.27	45	42.05 ± 0.52	49	42.03 ± 0.34	49	42.08 ± 0.27	49
Zephyr	SW	55.05 ± 0.54	61	210.02 ± 0.3	214	23.00 ± 0.00	23	22.01 ± 0.22	24
	TC	20.01 ± 0.19	23	23.00 ± 0.00	23	22.00 ± 0.00	22	22.01 ± 0.19	24
	LC	54.99 ± 0.19	55	52.02 ± 0.27	55	50.00 ± 0.00	50	50.02 ± 0.27	53
FreeRTOS	SW	181.03 ± 0.44	188	102.18 ± 1.64	119	84.99 ± 0.06	85	758.14 ± 1.37	773
	TC	82.01 ± 0.19	85	77.02 ± 0.32	82	76.04 ± 0.48	83	79.02 ± 0.32	84
	LC	91.12 ± 1.10	104	91.08 ± 0.49	94	95.01 ± 0.13	97	98.02 ± 0.20	101



(a) Send API latency.



(b) Receive API latency.

Fig. 11 Message queues API latency by message size

After each iteration of this process, the sending thread modified the message contents, and the receiving thread checked the content to ensure correctness. The results are depicted in Fig. 11, where the same experiment was repeated to multiple message sizes.

RIOT defines a structure to represent a message which contains a pointer to the data sent/received. However, unlike standard message queue implementations, the contents of this structure are copied from sending thread to the receiving thread, including the original pointer. Thus, giving direct access to the memory location of the original data to the receiving thread. Nonetheless, intending to keep agnosticism, the ChamelIoT framework uses the same mechanism of only copying the structure

contents on message queue operations. This results in identical results across the board, i.e., all three setups on both APIs, as the message size is always the structure size.

The TC setup on Zephyr shows consistent latency decreases for both *Send* and *Receive* APIs, averaging across all message sizes 65.80%, and 67.63%, respectively. On the LC configurations, up to message sizes of 16 words, the hardware setup performs better on both APIs. However, for bigger messages the latency is increased by 11.61% on *Sends* of 64 words and 10.76% on *Receives* of 64 words.

Finally, the TC configuration on FreeRTOS offers a consistent latency decrease to both functions on message sizes up to 16 words. For bigger messages, the latency decrease goes from an average of 45.33 to 16.92% on *Sends* of 64 words, and from an average of 44.09 to 14.31% on *Receives* of 64 words. The LC setup shows a similar behavior, for bigger message sizes, the performance is worse. This setup only offers latency decreases for messages up to four words on both *Send* and *Receive* functions. For bigger messages the latency increase reaches up to 135.25% on *Sends* and 145.82% on *Receives*.

On LC setups, the performance degradation observable for bigger message sizes can be justified by the fact that the memory operations are made through a TileLink DMA node, with lesser priority in the interconnects that are closer to the system memory. Unlike the TC setup that has direct access to the memory port.

5.3 Thread-metric benchmark suite

To evaluate the impact of using the ChamelIoT framework fully integrated into different IoT OSes, we use the Thread-Metric Benchmark Suite. This synthetic suite implements several benchmarks that stress a singular kernel service separately, allowing us to understand the influence of each service when used in run-time by an IoT application. Contrarily to the latency evaluation reported previously, this benchmark suite results take into consideration all other moving parts in the system, e.g., the tick system or context switching, despite a singular service being emphasized in each experiment. To test the services implemented in hardware by the ChamelIoT framework, we ran the following benchmarks:

1. Basic processing: a single thread performs mathematical operations in a loop.
2. Cooperative scheduling: five threads with the same priority execute concurrently, yielding in a loop.
3. Preemptive scheduling: five threads with increasing priorities, each resuming the next thread with a higher priority and suspending themselves in a loop.
4. Message processing: a single thread sends a message to itself through a message queue in a loop.
5. Synchronization: a single thread gives and takes a semaphore in a loop.

Thread-Metric benchmarks count the number of times each loop is repeated, presenting this count as a score after a period of time. For each experiment, the higher score means that the loop was executed more times. Thus there was less time wasted

Table 7 Thread metric benchmark results

		Basic processing	Cooperative scheduling	Preemptive scheduling	Message processing	Synchronization
RIOT	SW	67 902	4 014 564	1 989 564	2 154 840	7 116 828
	TC	67 902	5 868 623	2 863 945	9 204 934	20 295 651
	LC	67 902	5 399 884	1 968 619	4 372 335	8 897 766
Zephyr	SW	63 404	1 140 718	705 373	2 827 900	6 863 450
	TC	63 403	1 248 817	716 462	7 273 403	13 727 347
	LC	63 402	1 146 146	612 182	3 919 407	7 674 172
FreeR-TOS	SW	59 223	1 614 329	756 498	2 293 793	3 868 494
	TC	63 166	4 834 809	2 084 595	3 641 817	4 539 084
	LC	63 165	4 598 353	1 847 286	2 349 219	3 867 702

in the service under test, i.e., a higher count indicates better performance. Table 7 summarizes the results gathered from running each benchmark in periods of 30 s. The values presented correspond to the average of 100 samples for a specific benchmark. We performed the same experiments for the three configurations (SW, TC, and LC) on each OS (RIOT, Zephyr, and FreeRTOS).

RIOT performance is improved by both hardware setups. The TC configuration offers bigger performance increases than the LC setup, corroborating the results discussed in the previous section. For the Cooperative and Preemptive Scheduling benchmarks, TC setup improves the performance by 46.18% and 43.95%, respectively. The Message Processing tests show the best results for RIOT OS, where the TC improves the system by 327.17%. Despite the LC setup showing latency increases in some APIs, the performance is still better than the SW configuration. For instance, the LC setup increases the latency on semaphore operations. However, it presents a 25.02% performance increase. The same is true for the Message Processing benchmark, where the results obtained are 102.91% better.

The TC configuration on Zephyr improves the system performance up to 157.20% and 100.01% on the Message Processing and Synchronization benchmarks. For both Scheduling experiments, this configuration only offers small increments in the results. According to the latency evaluation on the LC setup, all the benchmarks tested should have shown worse performance than the SW approach. However, this is only true for the Preemptive Scheduling test. All the others have better results, particularly the Message Processing and Synchronization benchmarks, which improve the performance by 38.60% and 11.81%.

Lastly, the FreeRTOS is the only OS where the Basic Processing benchmark is improved by the hardware configurations. This happens because FreeRTOS scheduling is tick-based, and at every tick, the kernel executes and tries to schedule the next thread. Regarding the other benchmarks, both setups show better performance. Nonetheless, both greatly improve the performance on the Cooperative Scheduling benchmark (TC setup increase by 199.49% and LC setup by 184.85%) and on the Preemptive Scheduling benchmark (TC setup increase by 175.56% and LC setup by 144.19%).

5.4 Hardware resources

Measuring the FPGA resources consumed enables the developers to choose a platform that fits their needs. The FPGA size heavily influences the system form factor and power consumption, therefore being a major design choice on IoT devices. On the ChamelIoT framework, the most impacting factor on resource consumption is the total amount of threads supported. As mentioned previously, this number is proportional to the number of *Thread Nodes* on the Thread Manager and the number of bits used to identify each thread on the accelerator. Since the TID usage is propagated throughout the hardware, increasing the TID bit size results in a significant resource consumption increase.

Table 8 FPGA resource consumption for a system allowing 8 threads and 8 priorities

Module		Logic LUTS	LUTRAMS	SRLs	FFs	RAM blocks	DSP blocks
TC	E300ArtyDevKitFPGAChip	21191	868	60	14430	24	2
	ChamelIoT HW Accel.	5987	190	0	3726	0	0
	%	28.25%	21.89%	0.00%	25.82%	0.00%	0.00%
LC	E300ArtyDevKitFPGAChip	22679	728	60	15635	25	2
	ChamelIoT HW Accel.	7094	0	0	5156	1	0
	%	31.28%	0.00%	0.00%	32.98%	4.00%	0.00%

Table 9 FPGA resource consumption for a system allowing 16 threads and 16 priorities

Module		Logic LUTS	LUTRAMS	SRLs	FFs	RAM blocks	DSP blocks
TC	E300ArtyDevKitFPGAChip	34272	868	60	12655	24	2
	ChamelIoT HW Accel.	19767	190	0	2274	0	0
		57.68%	21.89%	0.00%	17.97%	0.00%	0.00%
LC	E300ArtyDevKitFPGAChip	40875	728	60	16399	25	2
	ChamelIoT HW Accel.	24388	0	0	5917	1	0
	%	59.66%	0.00%	0.00%	36.08%	4.00%	0.00%

Evaluating the impact of increasing the number of threads and priorities on the resources consumed by ChamelIoT was done extensively in previous works (Silva et al. 2022). However, the experiments only considered the tightly-coupled approach for the accelerator. To have a better understanding of how the two coupling approaches influence resource consumption. Tables 8 and 9 show the FPGA resource distribution for the two setups in different scenarios, one with a maximum of eight threads and eight unique priorities and the other with 16 threads and priorities. For both cases, the remaining system configurations were kept the same.

For the scenario with eight threads, albeit very close, the TC setup uses fewer Look-Up Tables (LUTs) than the LC setup. As depicted in Table 8, the TC setup uses fewer Logic LUTs, but on the other hand, it also uses LUTs as RAM (LUTRAMs), which the LC setup does not consume. The amount LUTs, including Shift-Register LUTs (SRLs), used by TC setup is 27.93% of the total amount of LUTs used by the whole system, identified as *E300ArtyDevKitFPGAChip* on the following tables. Regarding the Flip-Flop usage (FFs), the TC configuration also consumes less than the LC setup, amounting to 25.82% of the system total for the TC setup and 32.98% for the LC one. The resource distribution differences in both setups are justified by the different interfaces they have with the MCU and the required modifications to the Control Unit to accommodate these interfaces. Furthermore, the remaining hardware elements, namely the CPU and memory, also have to adapt to the accelerator coupling approach, consequently having minimal changes to the resource consumption.

The experiment with 16 threads shows upscaled consumption, as depicted in Table 9. The TC setup uses 56.70% of the system's total LUTs, and the LC setup uses 58.54%. This was the most impacted resource by the increasing number of threads and priorities. In particular, Logic LUTs was the only resource that significantly increased its consumption. This fact reflects the effect of incrementing the TID bit size, which extensively increases the logic across the entire accelerator.

5.5 Power consumption

Power consumption is a metric highly dependent on the application since it is the application dictates the time the processor spends in low-power or sleeping modes. Furthermore, the application controls which peripherals are active, consequently

Table 10 ChamelIoT power estimation

	Rocket	ChamelIoT-TC		ChamelIoT-LC	
		8T8P	16T16P	8T8P	16T16P
Static (W)	0.099	0.099	0.099	0.099	0.099
Dynamic (W)	0.196	0.201	0.218	0.212	0.24
%		2.55%	11.22%	8.16%	22.45%
Total (W)	0.295	0.3	0.317	0.311	0.339
%		1.69%	7.46%	5.42%	14.92%

impacting energy consumption. Taking this into consideration, to evaluate the effect of ChamelIoT on the system's power consumption, we opted to use the Power Analysis tools included in the Vivado Design Suite. The tool ran in vectorless mode with the default settings and power optimizations disabled and with the platform constraints for the Arty A7-100 board. The report includes dynamic power consumption, which is determined by the switching activity of clocks and datapaths, and static power consumption, which represents the minimum power consumption required to operate the hardware blocks.

Table 10 summarizes the results gathered for the following system configurations: (1) Rocket Core without ChamelIoT's hardware accelerator, (2) ChamelIoT tightly-coupled to the core, and (3) ChamelIoT loosely-coupled. Both setups with ChamelIoT were evaluated while supporting 8 and 16 threads, with a corresponding number of unique priorities.

The Rocket core deployed on the Arty A7-100T board presents an expected power dissipation of 0.295 W. When the hardware accelerator is included, there is an increase in the estimated dynamic power consumed while the static power consumption remains the same. The TC setup presents a power consumption increase of 1.69% for the 8-thread configuration and 7.46% for the 16-thread configuration. On the other hand, the LC setups show increases of 5.42% and 14.02% for the 8- and 16-thread configurations, respectively. For both TC and LC setups, the configuration with a higher amount of threads supported have higher power consumption, which is justified by increased hardware resource consumption presented in the previous section.

5.6 Memory footprint

The code size of an IoT application heavily influences the platform used in a system since a non-volatile memory needs to accommodate all the code. On low-end IoT devices, the system often uses the RAM available in the SoC for runtime needs and external memories to store the code. The memory footprint is a metric highly dependent on the application, as the code size and variables and arrays allocated will directly affect the memory used.

We used the Preemptive Scheduling benchmark from the Thread-Metric Benchmark Suite as the baseline for all three OSes to assess the memory footprint. This way, the application code was kept the same, with the exception of the kernel since it changes between OSes. Using the GNU RISC-V Toolchain, we gathered the amount of ROM and RAM required by each OS on all three setups, SW, TC, and LC, as depicted in Table 11.

Table 11 ChamelIoT memory footprint

	RIOT			Zephyr			FreeRTOS		
	SW	TC	LC	SW	TC	LC	SW	TC	LC
RAM (B)	8384	8384	8384	11,376	11,376	11,376	12,372	12,372	12,372
ROM (B)	19,346	18,220	18,742	22,448	21,668	22,040	22,994	21,944	22,200
%		5.82%	3.12%		3.47%	1.82%		4.57%	3.45%

Across all experiments, the amount of used RAM never changes because the ChamelIoT Abstraction Layer does not initialize any variables and mostly consists of simple RoCC instructions or memory accesses that replace code from the kernel internals. For this reason, setups using the ChamelIoT framework will have smaller code sizes.

RIOT presents a 5.82% decrease in the code size for the TC setup and 3.12% for the LC setup. Zephyr shows a decrease of 3.47% and 1.82% for the TC and LC setups, respectively. Moreover, FreeRTOS presents code size decreases of 4.57% on the TC setup and 3.45% on the LC setup. As mentioned earlier, regardless of OS, the LC setup requires more instructions on each function in the ChamelIoT Abstraction Layer. This is reflected in the memory footprint results since the TC setup shows better results for all three OSes.

6 Discussion

Software abstraction The ChamelIoT framework resorts to a set of macros, a minimalist abstraction layer, and slight modifications to each kernel to provide agnosticism for the application developer. This allows the use of hardware acceleration from an application by defining a variable at compile time without any other modifications to the code. The current abstraction layer follows the same logic for both approaches, where inputs need to be provided to the accelerator alongside an instruction opcode to execute any function. While this algorithm is optimal for the tightly-coupled approach, it fails to leverage the benefits of having multiple registers with data readily available on the loosely-coupled approach. We believe that with more optimizations to the software abstraction layer and minimal modification to the loosely-coupled accelerator, the increased API latency measured can be mitigated or even turned into a latency decrease. This could easily be achieved by having some fields from the *Thread Node* available as read-only registers, for instance. As such, it would decrease the number of instructions required to access a single value from four to one.

Additional features As mentioned before, some kernel services (e.g., time, memory, and interrupt management) were left aside from the current implementation. Notwithstanding, we strongly believe that they can be implemented as optional features to further enhance the OS performance. Furthermore, to achieve the goal of widespread adoption, more IoT OSes will be supported by the ChamelIoT framework. Additional services, like spin locks or mailboxes, or alternative services, e.g., different scheduling policies, will also be included on the hardware accelerator. We also believe that including more configurability points will help towards the goal of adoption. Lastly, a Configuration and Building Tool is a work in progress that aims to provide an easy-to-use tool that configures and builds the whole system stack, further minimizing the barrier of entry regarding hardware acceleration.

System evaluation The work presented only shows the evaluation of ChamelIoT from a synthetic standpoint and mostly encompasses performance and determinism experiments. We plan to extend the system evaluation to other fields like memory footprint and power consumption, as these are important metrics for any IoT application. Furthermore, we have a working setup of IEEE802.15.4 radio network where one of the nodes is a ChamelIoT accelerated system. This node uses RIOT with a

loosely-coupled ChamelIoT accelerator and implements a UDP server that receives and answers requests from other radios. Preliminary results show that the hardware accelerated configuration can process more messages per second, i.e., has better throughput, than the software implementation. We are working on extending the same setup to the other approach and remaining OSes to have better comparison and evaluation of the system in a real-world application.

Supporting additional IoT OSes Adapting ChamelIoT to support other IoT OSes is done by modifying the kernel internals to use ChamelIoT's Abstraction Layer wherever needed, as described in Sect. 4.2. However, some OSes may require specific functionalities to be added in the Hardware Accelerator to keep their behavior unmodified. In the ideal cases, the hardware modifications are limited to existing instructions, e.g., replacing the scheduling algorithm, which requires the developer to keep the same inputs and outputs. Otherwise, new instructions need to be added to support new features, in which case, the developer should implement these instructions following the existing examples on ChamelIoT and ensure they follow a similar approach regarding inputs, outputs, and how they interact with the software.

Porting ChamelIoT to other platforms Utilizing ChamelIoT's Hardware Accelerator on other Rocket-based platforms does not require modifications other than including the accelerator in the core. However, porting ChamelIoT to other RISC-V cores is a process that involves modifying the interfaces that connect the accelerator to the CPU and the memory system. Some RISC-V cores, like CV32E40P or PicoRV32, already have a dedicated custom interface for coprocessors, allowing the integration of tightly-coupled accelerators. In these cases, porting ChamelIoT-TC consists of removing the logic that interacts with RoCC and implementing new logic for the new interface while ensuring the remaining components keep the same behavior. On the other hand, porting the ChamelIoT-LC setup can be achieved by replacing the TileLink connection with interconnects of other peripheral buses, like AXI. Alternatively, when the target platform uses AXI, the developer can decide to add a TileLink-AXI adapter to the existing infrastructure instead of replacing the current interface.

7 Conclusions

In this paper, we presented ChamelIoT, an agnostic hardware OS framework for FPGA-based IoT devices. It leverages the advantage of an open-source ISA, RISC-V, to implement hardware acceleration for kernel services. Together with a minimalist software abstraction layer and slight modifications to the OS internals, ChamelIoT allows IoT applications to benefit from hardware acceleration without needing to modify the application code. This paper extends the previous by implementing the hardware acceleration in a different coupling approach where the accelerator is connected to the core through the memory interface, i.e., loosely-coupled to the MCU. The original coupling approach uses a coprocessor interface to integrate the accelerator in the MCU datapath, i.e., tightly-coupled.

We have deployed and evaluated our system with three different OSes: RIOT, FreeRTOS, and Zephyr. The experiments realized encompass API latency tests,

overall system performance measurement, and FPGA resource consumption. Results demonstrated that regardless of the coupling approach, hardware acceleration can be used to improve performance and determinism on IoT OSes. Furthermore, we can conclude that a tightly-coupled approach offers greater performance increases, reaching values of 199.49% for the overall system. Notwithstanding, the loosely-coupled configuration can more easily be adapted to other architectures, offering more in terms of scalability and re-usability.

Acknowledgements This work has been supported by FCT - *Fundação para a Ciência e Tecnologia* within the R &D Units Project Scope: UIDB/00319/2020 and SFRH/BD/146678/2019.

Funding Open access funding provided by FCT|FCCN (b-on).

Data availability The data that supports the findings of this study is available with the corresponding author, Miguel Silva, upon reasonable request.

References

Agron J, Peck W, Anderson E, Andrews D, Komp E, Sass R, Baijot F, Stevens J (2006) Run-time services for hybrid CPU/FPGA systems on chip. In: 2006 27th IEEE international real-time systems symposium (RTSS'06). pp 3–12

Alexandrescu A, Botezatu N, Lupu R (2022) Monitoring and processing of physiological and domotics parameters in an Internet of Things (IoT) assistive living environment. In: 2022 26th international conference on system theory, control and computing (ICSTCC). pp 362–367

Allied Market Research (2023) Row-end FPGA market by technology (EEPROM, antifuse, SRAM, flash, others), by node size (less than 28 nm, 28–90 nm, more than 90 nm), by application (telecommunication, automotive, industrial, consumer electronics, data center, medical, aerospace and defense, others): global opportunity analysis and industry forecast

ARM (2013) AMBA AXI and ACE protocol specification. Version E

Asanovic K, Patterson DA (2014) Instruction sets should be free: the case for RISC-V. Technical report, EECS Department, University of California, Berkeley

Asanović K, Avizienis R, Bachrach J, Beamer S, Biancolin D, Celio C, Cook H, Dabbelt D, Hauser J, Izraelevitz A, Karandikar S, Keller B, Kim D, Koenig J, Lee Y, Love E, Maas M, Magyar A, Mao H, Moreto M, Ou A, Patterson DA, Richards B, Schmidt C, Twigg S, Vo H, Waterman A (2016) The rocket chip generator. Number UCB/EECS-2016-17

Baum DR, Winget JM (1990) Real time radiosity through parallel processing and hardware acceleration. In: Proceedings of the 1990 symposium on interactive 3D graphics, I3D '90, New York, NY, USA. Association for Computing Machinery, pp 67–75

Boutros A, Nurvitadhi E, Ma R, Gribok S, Zhao Z, Hoe JC, Betz V, Langhammer M (2020) Beyond peak performance: comparing the real performance of AI-optimized FPGAS and GPUS. In: 2020 international conference on field-programmable technology (ICFPT). pp 10–19

Brebner G (1996) A virtual hardware operating system for the Xilinx XC6200. In: Hartenstein RW, Glesner M (eds) Field-programmable logic smart applications, new paradigms and compilers, Berlin, Heidelberg. Springer, Berlin, Heidelberg, pp 327–336

Cao K, Liu Y, Meng G, Sun Q (2020) An overview on edge computing research. IEEE Access 8:85714–85728
Chandra TB, Verma P, Dwivedi AK (2016) Operating systems for Internet of Things: a comparative study. In: Proceedings of the second international conference on information and communication technology for competitive strategies, ICTCS '16, New York, NY, USA. Association for Computing Machinery
Chéour R, Khriji S, El Houssaini D, Baklouti M, Abid M, Kanoun O (2019) Recent trends of FPGA used for low-power wireless sensor network. IEEE Aerosp Electron Syst Mag 34(10):28–38
Cunha L, Roriz R, Pinto S, Gomes T (2022) Hardware-accelerated data decoding and reconstruction for automotive LiDAR sensors. IEEE Trans Veh Technol 72:4267–4276
Davide Schiavone P, Conti F, Rossi D, Gautschi M, Pullini A, Flamand E, Benini L (2017) Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for internet-of-things applications. In: 2017 27th international symposium on power and timing modeling, optimization and simulation (PATMOS). pp 1–8
Dietrich C, Lohmann D (2017) OSEK-V: application-specific RTOS instantiation in hardware, vol 52. Association for Computing Machinery, New York, pp 111–120
DivyaKrishna K, Akkala V, Bharath R, Rajalakshmi P, Mohammed AM, Merchant SN, Desai UB (2016) Computer aided abnormality detection for kidney on FPGA based IoT enabled portable ultrasound imaging system. IRBM 37(4):189–197
Engel A, Koch A (2016) Heterogeneous wireless sensor nodes that target the Internet of Things. IEEE Micro 36(6):8–15
Fritzmann T, Sigl G, Sepúlveda J (2020) RISQ-V: tightly coupled RISC-V accelerators for post-quantum cryptography. IACR Trans Cryptogr Hardw Embed Syst 2020(4):239–280
Gomes T, Garcia P, Salgado F, Monteiro J, Ekpanyapong M, Tavares A (2015) Task-aware interrupt controller: priority space unification in real-time systems. IEEE Embed Syst Lett 7(1):27–30
Gomes T, Garcia P, Pinto S, Monteiro J, Tavares A (2016) Bringing hardware multithreading to the real-time domain. IEEE Embed Syst Lett 8(1):2–5
Gomes T, Salgado F, Tavares A, Cabral J (2017) Cute mote, a customizable and trustable end-device for the Internet of Things. IEEE Sens J 17(20):6816–6824
Gomes T, Sousa P, Silva M, Ekpanyapong M, Pinto S (2022) FAC-V: an FPGA-based AES coprocessor for RISC-V. J Low Power Electron Appl 12(4):50
Hahm O, Baccelli E, Petersen H, Tsiftes N (2016) Operating systems for low-end devices in the Internet of Things: a survey. IEEE IoT J 3(5):720–734
Hofer W, Lohmann D, Scheler F, Schröder-Preikschat W (2009) Sloth: threads as interrupts. In: 2009 30th IEEE real-time systems symposium. pp 204–213
Iturbe X, Benkrid K, Hong C, Ebrahim A, Torrego R, Arslan T (2015) Microkernel architecture and hardware abstraction layer of a reliable reconfigurable real-time operating system (R3TOS). ACM Trans Reconfigurable Technol Syst 8(1):1–35
Johnson AP, Chakraborty RS, Mukhopadhyay D (2015) A PUF-enabled secure architecture for FPGA-based IoT applications. IEEE Trans Multi-Scale Comput Syst 1(2):110–122
Karray F, Jmal MW, Garcia-Ortiz A, Abid M, Obeid AM (2018) A comprehensive survey on wireless sensor node hardware platforms. Comput Netw 144:89–110
Lange AB, Andersen KH, Schultz UP, Sørensen AS (2012) HartOS—a hardware implemented RTOS for hard real-time applications. In: 11th IFAC, IEEE international conference on programmable devices and embedded systems, vol 45. pp 207–213
Lübbers E, Platzner M (2009) ReconOS: multithreaded programming for reconfigurable computers. ACM Trans Embed Comput Syst 9(1):1–33
Maruyama N, Ishihara T, Yasuura H (2010) An RTOS in hardware for energy efficient software-based TCP/IP processing. In: 2010 IEEE 8th symposium on application specific processors (SASP). pp 58–63
Maruyama N, Ishikawa T, Honda S, Takada H, Suzuki K (2014) ARM-based SoC with loosely coupled type hardware RTOS for industrial network systems. Proc. Operating Systems Platforms for Embedded Real-Time applications (OSPERT'14). pp 9–16
Najafi M, Zhang K, Sadoghi M, Jacobsen H-A (2017) Hardware acceleration landscape for distributed real-time analytics: virtues and limitations. In: 2017 IEEE 37th international conference on distributed computing systems (ICDCS). pp 1938–1948
Oliveira ASR, Almeida L, de Brito Ferrari A (2011) The ARPA-MT embedded SMT processor and its RTOS hardware accelerator. IEEE Trans Ind Electron 58(3):890–904
Oliveira D, Costa M, Pinto S, Gomes T (2020) The future of low-end motes in the Internet of Things: a prospective paper. Electronics 9(1):111

Ong SE, Lee SC, Ali Noohul BZ, Hussin Fawnizu AB (2013) SEOS: hardware implementation of real-time operating system for adaptability. In: 2013 first international symposium on computing and networking. pp 612–616

Pala D (2017) Design and programming of a coprocessor for a RISC-V architecture. Master's Thesis, Collegio di Ingegneria Informatica, del Cinema e Meccatronica

Pena MDV, Rodriguez-Andina JJ, Manic M (2017) The Internet of Things: the role of reconfigurable platforms. IEEE Ind Electron Mag 11(3):6–19

Perera C, Liu CH, Jayawardena S, Chen M (2014) A survey on Internet of Things from industrial market perspective. IEEE Access 2:1660–1679

Pinto S, Cabral J, Gomes T (2017) We-care: an IoT-based health care system for elderly people. In: 2017 IEEE international conference on industrial technology (ICIT). pp 1378–1383

Qiu J, Wang J, Yao S, Guo K, Li B, Zhou E, Yu J, Tang T, Xu N, Song S, Wang Y, Yang H (2016) Going deeper with embedded FPGA platform for convolutional neural network. In: Proceedings of the 2016 ACM/SIGDA international symposium on field-programmable gate arrays, FPGA '16, New York, NY, USA. Association for Computing Machinery, pp 26–35

Sá B, Martins J, Pinto S (2022) A first look at RISC-V virtualization from an embedded systems perspective. IEEE Trans Comput 71(9):2177–2190

Sanchez-Iborra R, Cano M-D (2016) State of the art in LP-WAN solutions for industrial IoT services. Sensors 16(5):708

SEMICO Research Corporation (2019) RISC-V market analysis the new kid on the block. Technical report, Semico

SiFive (2018) SiFive TileLink Specication. Version 1.8.1

Silva M, Cerdeira D, Pinto S, Gomes T (2019) Operating systems for Internet of Things low-end devices: analysis and benchmarking. IEEE Internet Things J 6(6):10375–10383

Silva M, Gomes T, Pinto S (2022) Agnostic hardware-accelerated operating system for low-end IoT. pp 21–30

Waterman AS (2016) Design of the RISC-V instruction set architecture. PhD Thesis, UC Berkeley

Wei Y, Liang F, He X, Hatcher WG, Chao L, Lin J, Yang X (2018) A survey on the edge computing for the Internet of Things. IEEE Access 6:6900–6919

Zhang X, Ramachandran A, Zhuge C, He D, Zuo W, Cheng Z, Rupnow K, Chen D (2017) Machine learning on FPGAS to face the IoT revolution. In: 2017 IEEE/ACM international conference on computer-aided design (ICCAD). pp 819–826

Zhao L, Machado Matsuo IB, Zhou Y, Lee W-J (2019) Design of an industrial IoT-based monitoring system for power substations. pp 1–6

Zikria YB, Heejung Yu, Afzal MK, Rehmani MH, Hahm O (2018) Internet of Things (IoT): operating system, applications and protocols design, and validation techniques. Future Gener Comput Syst 88:699–706

Miguel Silva is a Ph.D. candidate at the University of Minho. During his master's thesis at University of Minho, he worked in the development and testing of automotive instrument clusters for a major industry company, and late he joined a research project related with video and multimedia. In these projects, he refined his knowledge of embedded systems, system software, and connectivity on low-end devices. At the moment, his focus is the development of an agnostic operating system in hardware for the Internet of Things.

Tiago Gomes has received the master's degree in telecommunications engineering and Ph.D. degree in electronics and computers engineering from the University of Minho, Braga, Portugal. He is a Research Scientist and Assistant Professor with the University of Minho. His current research interests include embedded hardware acceleration for autonomous perception systems based on LiDAR sensors, and hardware/software co-design for resource constrained Internet of Things devices.

Mongkol Ekpanyapong received the B.Eng. degree from Chulalongkorn University, the M.Eng. degree from Asian Institute of Technology (AIT), in, and the M.Sc. and Ph.D. degrees from Georgia Institute of Technology. He was a System Engineer with United Communication Network, and later he was a Senior Computer Architect with the Core 2 Architecture Design Team, Intel Corporation. Currently he is an Associate Professor in the School of Engineering and Technology, AIT. His research interests include microarchitecture, embedded systems, mechatronics, deep learning, and computer vision.

Adriano Tavares is an Associate Professor at University of Minho, Portugal. He holds a Ph.D. in Industrial Electronics from University of Minho, a Master of Science in Information Technology and an undergraduate degree in Informatics both from University of Coimbra. His research interests are embedded systems modeling and design, system software design, system-on-chip design and engineering education. He published multiple book chapters and papers on international conferences and journals related to embedded systems and two books on assembly and C programming.

Sandro Pinto is an Associate Research Professor at the Centro ALGORITMI, Universidade do Minho, Portugal. He holds a Ph.D. in Electronics and Computer Engineering. Dr. Sandro has a deep academic background and several years of industry collaboration focusing on operating systems, virtualization, and security for embedded, CPS, and IoT systems. He has published 100+ peer-reviewed papers and is a skilled presenter with speaking experience in top-tier academic and industrial conferences.

Authors and Affiliations

Miguel Silva[1] · Tiago Gomes[1] · Mongkol Ekpanyapong[2] · Adriano Tavares[1] · Sandro Pinto[1]

✉ Miguel Silva
miguel.silva@dei.uminho.pt

Tiago Gomes
mr.gomes@dei.uminho.pt

Mongkol Ekpanyapong
mongkol@ait.ac.th

Adriano Tavares
atavares@dei.uminho.pt

Sandro Pinto
sandro.pinto@dei.uminho.pt

[1] Centro ALGORITMI/LASI, Universidade do Minho, Guimaraes, Portugal

[2] Department of Industrial Systems Engineering, Asian Institute of Technology, Khlong Luang, Pathum Thani, Thailand