Agnostic Hardware-Accelerated Operating System for Low-End IoT

Miguel Silva, Tiago Gomes, and Sandro Pinto
Centro ALGORITMI, Universidade do Minho, Portugal

***Abstract*—There is increasing pressure to optimize Internet of things (IoT) low-end devices. The ever-growing number of requirements and constraints is pushing towards maximizing performance and real-time, but simultaneously minimizing power consumption, form factor, and memory footprint. This has motivated the adoption of Field-Programmable Gate Array (FPGA) technology to accelerate computing-intensive workloads in hardware. However, and despite the ongoing trend of migrating application-level tasks to hardware, recently, the offload of system software such as operating system (OS) services has received little attention. This paper presents CHAMELIoT, a framework for FPGA-based IoT platforms that provides agnostic hardware acceleration to OS services by leveraging RISC-V technology. CHAMELIoT allows for developers to run unmodified applications in a set of well-established IoT OSes. Currently, the framework has support for RIOT, Zephyr, and FreeRTOS. The evaluation showed that latency and determinism can be enhanced up to 10x while the system's performance can be increased to nearly 200%. CHAMELIoT will be open-sourced.**

I. INTRODUCTION

The growing popularity of the Internet of Things (IoT) is fostering the shift of computing workloads from remote centralized facilities to the edge. However, edge devices are often resource-constrained, which imposes some challenges while handling workloads intended for high-end devices [1]. Such devices are widely present in a broad range of sectors, e.g., transportation, health-care, or industrial, and are required to constantly gather and process data in applications that demand real-time and determinism on top of high-performance ratios. Fulfilling the different requirements that are dictated by the target application requires each final solution to be individually tailored to fit tight constraints [2, 3].

IoT endpoints have been outgrowing the capabilities of traditional microcontroller units (MCUs) to the extent that the industry is starting to adopt reconfigurable platforms to achieve the desired metrics [4]. Reconfigurable technology, namely Field Programmable Gate Array (FPGA), enables the development of custom accelerators by offloading compute-intensive tasks to hardware, often connected to the MCU as standard peripherals [5]. Until recent years, FPGAs presented several problems that impeded their mass adoption in IoT applications, such as high power consumption, large form factor, and difficulty of integration. However, platforms like embedded FPGAs or low-power FPGAs, are already minimizing those issues and thus, seeing increasing applicability in low-end IoT devices [6]. Nevertheless, deploying and optimizing accelerators on FPGA has been hampered by several challenges, which start to be alleviated with the rise of RISC-V.

RISC-V is a novel open-source instruction set architecture (ISA) that follows a reduced instruction set computer (RISC) design that enables the combination of extensions that eases the integration of dedicated hardware accelerators with the application software [7, 8]. Some RISC-V implementations, such as the Rocket chip [9], already extend the ISA by defining a subset of instructions for user-defined co-processors, while other implementations expect these co-processors to be managed through a memory-mapped interface. Tightly- and loosely-coupled approaches pose different challenges and present different trade-offs regarding performance, determinism, real-time, system integration, and portability [10].

Low-end IoT devices often feature embedded real-time operating systems (RTOS) to support the desired end application. RTOSes enforce scheduling policies, implement synchronization (e.g., mutexes) and communication (e.g., message queues) mechanisms, and provide abstractions from the hardware [11, 12]. Aiming at improving the determinism and performance of RTOSes, some kernel services have been migrated to hardware. So far, this trend has been facing several challenges mainly due to the lack of reconfigurable hardware and the barriers imposed by closed computer architectures [10]. With the emergence of RISC-V alongside reconfigurable low-end platforms [13, 14], there is a renewed opportunity to explore the migration of kernel services to hardware.

This paper presents CHAMELIoT, a framework for reconfigurable IoT devices that aims at providing agnostic hardware acceleration for different OS kernel services. Taking advantage of the RISC-V ISA extensions, it is possible to integrate dedicated hardware accelerators without adding complex software abstraction layers. The developed accelerators can be used transparently by any RTOS, bringing the benefits of hardware acceleration while keeping the application unmodified. The evaluation showed that latency and determinism can be enhanced up to 10x while the system's performance is increased to nearly 200%.

The main contributions of this paper are:

1) An open-source agnostic OS framework that supports a variety of RTOSes, requiring few changes to the kernel while keeping the end-user interface intact / unmodified;
2) Highly configurable hardware kernel modules (e.g., scheduling, thread management, synchronization, and communication mechanisms) leveraging RISC-V technology and adopting different coupling approaches;
3) Deployment and benchmarking of three widely-used OSes for IoT low-end systems: Zephyr, RIOT, and FreeRTOS.

II. ChamelIoT Overview

A. Motivation

Offloading OS kernel services to hardware have been a particular topic of interest for the academy [10, 15–22]. Despite the considerable amount of attempts reported in the literature, both reconfigurable and hardware-accelerated OSes have not gained traction in the industry, due to several limitations that have hampered their adoption. Below, we highlight the main reasons that have contributed to that lack of adoption.

Software Interface. Hardware OSes typically provide custom APIs, which impose steep learning curves and enlarge the desired (industry target) time-to-market. To address these issues, some works use compatibility standards like POSIX [17] while others solutions provide generic APIs that easily map to standard RTOS interfaces [10, 20, 21]. Nonetheless, these two approaches typically have their intrinsic limitations, because not all legacy applications leverage POSIX standards nor the process to map generic to standard RTOS APIs is automated.

Target Architecture. Hardware OSes are usually developed targeting a single CPU or architecture, limiting their overall usability and hindering their adoption. Implementing a tighlty-coupled accelerator requires modifying the CPU, which, for proprietary ISA like Arm, is often impossible due to violation of intellectual property. Open ISAs, such as RISC-V, are now offering the opportunity to support tightly-coupled acceleration via co-processors or ISA extensions.

Application Suitability. Due to the IoT heterogeneity, each application has different requirements and constraints. To cope with this heterogeneity, hardware OSes must provide enough configurability points to not only fit the hardware constraints but also deploy hardware modules only when needed. This is typically not available on hardware RTOSes, which are usually tailored to fit a specific application.

With our work, we aim at overcoming the main challenges which have been hindering the adoption of hardware OSes, while offering the benefits of accelerating kernel services in hardware. ChamelIoT intends to contribute to the state-of-the-art with:

- An API that can be easily portable to multiple IoT OSes by mapping and replacing calls to the kernel internals to our framework interface (this process will be later automated by a configuration and building tool).
- A set of open hardware building blocks leveraging an open ISA (RISC-V), which enables easy integration of both tightly- and loosely-coupled accelerators.
- A set of open-source hardware OS services with multiple configurability points to ensure that the design variability of different OSes are fully captured and application's requirements and constraints are met.

B. Architecture

Figure 1 illustrates the architecture of the proposed framework. Built on top of RISC-V technology, the framework is composed of three main components: (1) tightly- and loosely-coupled Hardware Accelerator(s); (2) the Abstraction Layer; and (3) the Configuration and Building tool.

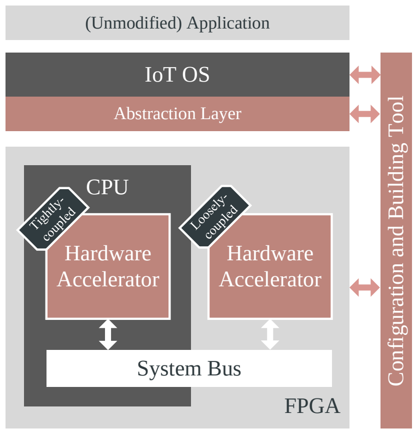

Fig. 1: Framework architecture.

Hardware Accelerator. One of the main goals of our framework is the acceleration of OS kernel services in hardware, such as the scheduler, thread manager, and synchronization and inter-process communication (IPC) mechanisms. By providing enough configurability to each service, 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- or tightly-coupled approach, which translates into better adaptability or better performance, respectively.

Abstraction Layer. The software API can be easily adapted and ported to most IoT OSes and RTOSes. Each service implemented in hardware has a set of minimalist interfaces to enable a fine-grained abstraction layer, and thus, allows for easy integration with software RTOSes. Additionally, a collection of abstractions is provided to enable the software to access and gather data from the hardware accelerator.

Configuration and Building tool. In order to ease the development process and learning curves, our framework intends to offer an external tool that can be used for hardware and software customization through a graphical user interface. This tool reduces the required knowledge about implementation details, simplifying the process of creating a solution with different hardware and software components.

C. Goals

The main goals of ChamelIoT Hardware Accelerator are:

Real-Time and Determinism. ChamelIoT must provide hard real-time guarantees and bounded worst-case execution time (WCET). Moreover, predictability shall not be affected by the number of active features on a certain component, e.g., the number of active threads.

Performance. ChamelIoT must improve system performance, by reducing the performance overhead introduced by standard RTOS services.

Flexibility. The framework must provide several configurability points to customize the ChameIIoT to the application requirements and constraints.

Agnosticism. The software interface must allow for the user to transparently run unmodified applications without awareness of the configuration of the system (software vs hardware).

D. Scope

The ChameIIoT is close to feature-complete, supporting at the moment the scheduling services, thread management, and synchronization and IPC mechanisms. The work presented in this paper refers to the Hardware Accelerator component of the framework, which is deployed as a tightly-coupled co-processor in a RISC-V core. In the short term, we intend to explore the trade-offs between tightly- and loosely-coupled configurations. Moreover, some OS services that have been already deployed in hardware by related work are not within the scope of our framework. These include:

Interrupt Management. The most common approach to manage interrupts is to trap and process them in a custom hardware accelerator. While this approach has proven to bring benefits, e.g., by removing multiple priority spaces [23, 24], it would require the developer to be aware of several implementation details which conflicts with the ChameIIoT's agnosticism.

Time Management. Most IoT OSes rely on platform-available timers to perform time management, e.g., system tick or managing delays and events. Migrating this time-related operations to hardware would require the replication a the timer logic in the FPGA fabric [20, 21], resulting in some sort of redundancy. Furthermore, it would also require redirecting the timer interrupt to a different source, compromising agnosticism.

Context Switching. As the most architecture-dependent feature, implementing the context switching in hardware would require extensive modifications to the CPU datapath. Despite the migration of this feature to hardware has proven to bring performance and determinism benefits [10], a highly-tailored core limits its flexibility and adaptability, and consequently its reusability and adoption.

Memory Management. Implementing heap and thread stacks management in hardware would require a large amount of resources. Since most FPGA-based IoT devices have limited logic elements, we will keep this service in software.

III. Hardware-accelerated Operating System

Given the heterogeneity of IoT applications, a myriad of IoT OSes have emerged to cope with the broad variety of requirements and constraints [11, 12]. However, their design choices, such as kernel architecture, scheduling policy, and other available features, can have a significant impact on the overall system's behavior and performance. Among the available OSes suitable for low-end IoT devices, we provide support to RIOT, Zephyr, and FreeRTOS. Such OSes present enough variability regarding the main design points we consider essential to implement and evaluate with our framework. Furthermore, they present extensive popularity and applicability in IoT applications, backed by continuous support from respective open-source communities.

TABLE I: Key features of each 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

RIOT, Zephyr, and FreeRTOS share similar design principles, e.g., they are based on a preemptive priority-based scheduler and implement a multi-queue thread[1] management system. However, there are still some design choices that have a major impact on the accelerator design, as summarized in Table I. Considering the resource limitations of low-end reconfigurable platforms, the Hardware Accelerator provides multiple configurability points to trade-off the number of supported features with the hardware. It is possible to configure the maximum number of threads, priority, mutexes, semaphores, and message queues, or finer-grained configurations such as the presence of priority inheritance in mutexes or the message size limit for message queues.

The current implementation of 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 (one of them can be used to interface custom accelerators).

The Hardware Accelerator is implemented tightly coupled to the E31 core by leveraging the Rocket Custom Co-processor (RoCC) interface. The RoCC interface is further divided into two sub-interfaces: (1) the *command interface*, which manages communication between the CPU and the co-processor, and (2) the *memory interface*, which provides the co-processor access to the memory system. 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 (Figure 2). It specifies the target co-processor, the source and destination of data, and the performing operation.

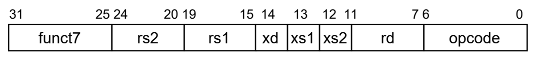

Fig. 2: RoCC instruction.

[1]FreeRTOS naming convention uses the term *Task* instead of *Thread*. In this manuscript, we use the term *Thread* to refer to *Task*.

The *opcode* is used to identify the co-processor, and per the RoCC specification, this field can only contain one of four predefined values, thus, limiting the number of co-processors to a maximum of four. The fields *rd, rs1, and rs2* specify the destination (rd) and source (rs1 and rs2) CPU registers used to transfer data with the co-processor. The *xd, xs1, and xs2* are used to identify which of the previous registers must be used by the instruction. Lastly, the field *funct7* is used as a user-defined opcode for each co-processor that indicates which function has to be executed. This instruction limits the interaction between the CPU and the co-processor to: (1) two 32-bit words being received on the co-processor; (2) a single 32-bit word response; and (3) a maximum of 128 distinct operations. Each of these instructions is implemented in software as inline functions to avoid the overhead of calling a function (prologue, epilogue, and jumps). Using macros, each API is then included in the three OS internals, while some sections of original code have been removed, or slightly adapted, without modifying its behavior. Most modifications are done at the lowest level, on leaf functions, therefore maintaining the execution flow of each OS.

Taking into account the variability and requirements of the target OSes, the RoCC interface, and the limitations of the instruction format, the Hardware Accelerator follows the architecture depicted by Figure 3. The two RoCC sub-interfaces are managed by the *Control Unit*, which is responsible for managing the commands sent/received to/from the CPU and managing all other components. The *Thread Manager* stores and manages the data related to each thread present on the system, while the remaining components, *Scheduler, Mutexes, Semaphores, and Message Queues*, are instantiated in arrays to implement the respective services. Each of these components is further detailed in the following sections.

A. Control Unit

The Control Unit is the main component of the Hardware Accelerator and is responsible for managing the RoCC interface (including the command and memory sub-interfaces) and the remaining hardware components to handle software requests. Whenever an instruction is issued by the core to the accelerator, the Control Unit reads the *funct7* to decode the operation requested and to identify the target component/service that matches the request. The input data, available in *rs1 and rs2* fields, is forwarded to the requested service, and the output data is returned in the *rd* field. Given the RoCC instruction protocol, the output has to be available at the same time the instruction is issued to the accelerator, requiring the Control Unit to be fully implemented with combinational logic (the same is valid for the remaining hardware components).

The Control Unit is also responsible for managing the multiple arrays of Mutexes, Semaphores, and Message Queues. The array index is selected per the ID from the instruction, and the maximum number of elements is set at compile-time. Since the RoCC interface only allows for one memory sub-interface, the Control Unit is also responsible to control all memory operations, despite the existence of multiple message queues that may require memory access. Additionally, message queues require local memory to store pending messages. This memory is also managed by the Control Unit since some message queue operations still require direct data transfers between the CPU and the message queue memories.

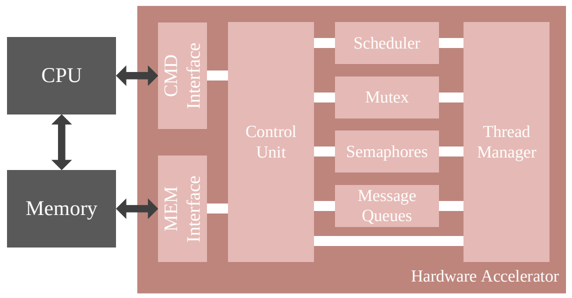

Fig. 3: Hardware accelerator architecture.

B. Thread Manager

The Thread Manager is mainly responsible for storing and managing the thread's data. This information is stored in a vector of *Thread Nodes*. Figure 4 depicts the *Thread Node* structure. The size of this vector fixes the maximum number of threads in the system (configured by the user) and the element's index in the array is used as the Thread Identifier (TID). Each *Thread Node* has a *data* field that stores the pointer to the Thread Control Block (TCB), provided by the OS. The hardware accelerator being capable of interchangeably operate with both TID and TCB offers more flexibility to accommodate several OS internals. To add a thread, it is necessary to provide the TCB pointer and thread's priority to the Thread Manager. Then, the Thread Manager checks the *dirty* bit of all nodes in the array to find one empty position in the array. If a position is available, the dirty bit is set and the corresponding index is returned to the software. In contrast, when the software requests a thread to be removed, the Thread Manager clears the corresponding dirty bit and a new index becomes available. The remaining fields in a Thread Node are used by the Thread Manager to control a multi-queue system (for threads that participate in the scheduling process). The multi-queue system includes a linked list per thread priority level (ready queue). Each linked list includes its respective threads that are in the ready or the running state. Therefore, the *priority* identifies the ready queue where the thread belongs, the *state* indicates the thread state (modifying it will cause the thread to be added or removed from the ready queue), and the *next* is the pointer to the next TID in the linked list.

Considering the timing constraints imposed by the RoCC protocol and to overcome the iterative flow of linked lists, the Thread Manager keeps track and manages the root, next,

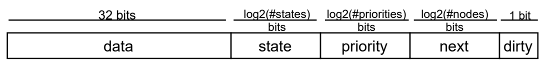

Fig. 4: Thread Node structure.

and last node of each list. With that, the Thread Manager is aware of each ready queue state, and is capable of adding and removing threads to the corresponding linked list in a single clock cycle. With the purpose of simplifying the scheduling process, each queue is implemented with a circular linked list. Since the Thread Manager is responsible for managing the ready queues and the thread data, it must also comply to the requests issued by other hardware components. The Scheduler accesses the Thread Manager to get the TIDs for the threads that will be scheduled, while the accesses from Mutexes, Semaphores, and Message Queues are to request threads to be added or removed from the ready queues.

C. 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. Thus, the thread with the highest priority in ready state will run until it yields its execution or it is preempted by a thread with higher priority. In case of multiple threads with the same priority, the scheduling policy follows a round-robin scheme to determine which thread is next to execute. To schedule the next thread, the Scheduler accesses the Thread Manager's ready queues to identify which linked lists are not empty, and which of them have the highest priority. The Scheduler is responsible for changing the current active thread state from *running* to *ready*, and the way around for the new thread. To maximize the timing guarantees (i.e., ensure determinism and minimize jitter), the Scheduler is implemented in combinational logic.

D. Mutexes

A mutex is a synchronization primitive that ensures mutually exclusive access to a resource. Whenever a thread *locks* a mutex, no other thread can lock the same mutex until the original owner *unlocks* it. For a thread to successfully lock a mutex, it must be unlocked, otherwise, the thread trying to lock it is changed to a blocked state and removed from the ready queue until the mutex owner unlocks it. Notwithstanding, whenever a thread is blocked, a priority inversion may occur resulting, in the worst case scenario, in a thread being permanently blocked. To solve this problem, some OSes implement a priority inheritance mechanism that raises the priority of the mutex owner thread when a thread with higher priority tries to lock a mutex.

Each Mutex implementation maintains a register with the current thread that owns the mutex and a list of TIDs of each thread that has been blocked trying to lock it. This list of threads also contains their respective priority, to provide a priority inheritance mechanisms. Whenever a thread's priority is raised, the Thread Manager removes that thread from the current ready queue and adds it to the inherited priority queue. The reverse operation is requested when the owner thread releases the mutex. Finally, whenever a thread is forced to change state, the scheduler is also updated accordingly and the software receives a response via RoCC instruction to issue a context switch.

E. Semaphores

Counting semaphores are synchronization mechanisms where a resource is produced by a thread and consumed by another. A producer thread signals the semaphore whenever the processed resource is ready, while a consumer thread checks if there are resources to be used, waiting otherwise. A semaphore allows for multiple resources to be shared among multiple threads. Each hardware Semaphore has an internal register to count the number of producer threads that have triggered a *give* operation. While this value is greater than zero, any *take* request from a consumer thread do not block the thread. However, once the count register is zero, the consumer thread is blocked and its TID stored into an internal list.

F. Message Queues

Message queues are asynchronous communication mechanisms between threads, which use a FIFO to store and share messages. Each message is a structure that contains the message size and its content, accessed through a pointer with a well-defined size. Whenever a thread *puts* a message in the queue, if there are no threads waiting for it, its content must be stored internally to be accessed when requested.

While the internal memory (where each waiting message is stored) is managed by the Control Unit, each Message Queue keeps track of which threads are trying to send a message that has yet been received, and which threads are waiting for a message. For the latter list of threads, each Message Queue also stores the pointer to where the thread intends to receive the data, so it can trigger the Control Unit to transfer the message from the internal memory to the CPU memory. Considering that all memory operations are performed sequentially and the RoCC standard implies a valid response when an instruction is issued, the data transfer between both memories is done in background, after the RoCC instruction is handled by the accelerator. During an ongoing transfer, the accelerator is not allowed to accept other Message Queue operations that require memory access, issuing an error message stating the Message Queue is busy.

IV. Evaluation

To evaluate the ChamelIoT framework, we have integrated and provide support to three IoT OSes: RIOT, FreeRTOS, and Zephyr. To assess performance and determinism, we measured the latency of most kernel services APIs (microbenchmark) by measuring the clock cycles required by each function. We also evaluated the overall system's performance using the Thread Metric benchmark suite, which provides a set of tests stressing the majority of RTOS-related features (e.g., scheduling, synchronization, IPC, etc.). Each experiment was performed for the three OSes targeting two different hardware configurations: (i) without using the hardware accelerator, i.e., the standard software implementation (henceforth referred as *SW*), and (ii) with support of hardware acceleration (hereafter referred as *HW*). Lastly, we evaluated the impact of ChamelIoT on the hardware resources required by different thread and priorities configurations, which (from empirical experiments) are the most impactful configurability points.

A. 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 connected through the RoCC Interface to an E31 Coreplex RISC-V core (RV32-IMAC). Both 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, 4 different mutexes, semaphores, and messages queues (each with 16-word size, and a queue of four messages). Regarding the resource consumption experiments, the aforementioned configurations were modified independently.

B. API Latency

To assess determinism and performance, we have measured the number of clock cycles required to execute the most common RTOS services, for both SW and HW configurations. Each experiment was repeated 10000 times, and for each kernel service, the results are presented by the average number of cycles along with the variance measured across all repetitions. The results are discussed as follows:

1) Scheduling and Thread Management: Figure 5 depicts the latency of three different APIs related to the scheduling and thread management: *schedule*, *thread suspend*, and *thread resume*. The first API occurs at every scheduling point, to schedule the next thread to execute. The last two APIs are responsible for changing the thread state, i.e., from ready to suspended state in *thread suspend*, and vice versa for the *thread resume*. To test the *schedule* service, we had two threads yielding execution, while measuring the clock cycles required to schedule the next thread. For the *thread suspend* API we had one higher priority thread suspending itself and for the *thread resume* API we had a lower priority thread resuming the previous thread execution.

For the SW system configuration, FreeRTOS presents higher execution times when compared to the other OSes. This is mainly due to the FreeRTOS kernel design related to managing lists, in particular, due to the need of moving threads between two different linked lists, e.g., from a ready queue to a suspended list. For the FreeRTOS HW setup, the latency of these APIs improved 819.1% in the *schedule*, 175.2% in the *thread suspend*, and 127.7% in the *thread resume*.

Zephyr has the most optimized scheduling algorithm, thus presenting improvements of 100.2% for the HW system setup. The *suspend* and *resume* APIs had marginal improvements of 9.6% and 9.3%, due to the amount of sanity checks performed

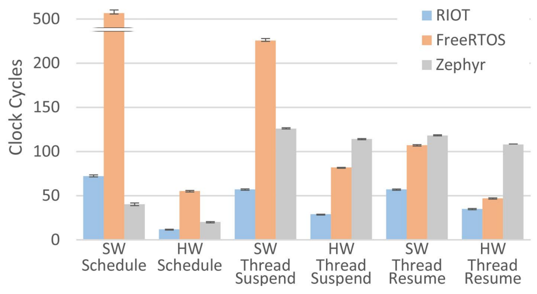

Fig. 5: Thread Management API latency.

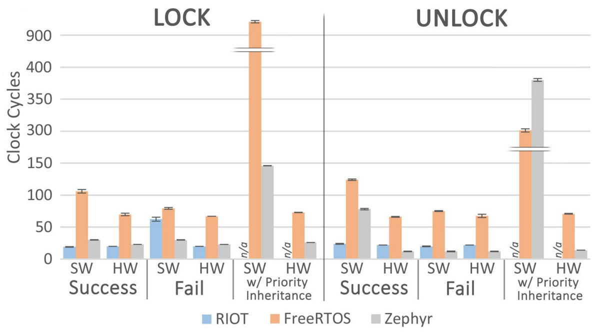

Fig. 6: Mutex APIs latency.

by the kernel in these APIs. In the HW setup, these sanity checks are translated into instructions to communicate with the accelerator to check the metadata related to the threads. Hence, the time spent for sanity checking in the HW configuration is similar to the one for the SW configuration.

Unlike Zephyr, RIOT performs almost no sanity check in these three services, resulting in significant gains in terms of performance, particularly for the *schedule* operation. For the HW setup, the latency is lowered 50.1% for the *schedule* API, and 9.4% and 8.5% for the *suspend* and *resume*, respectively.

2) Mutexes: There are two distinct APIs related to mutexes: *lock* and *unlock*. For both services we have evaluated three different behaviours: (1) first, the thread can successfully lock and unlock a mutex, in a loop; (2) second, the thread fails to lock a mutex, i.e., not being its owner, and fails to unlock a mutex (the mutex is already free). (3) third, a thread with lower priority locks a mutex and yields its execution to a higher priority thread that also attempts to lock the same mutex, triggering priority inheritance. Once the lower priority thread reclaims execution, it unlocks the mutex, reverting the priority changes. The results gathered for both APIs in the three different OSes are shown in Figure 6.

RIOT does not show any improvement regarding the latency of successful *locks* and both successful and failed *unlocks*. For these experiments, for the SW setup, RIOT presents already a small standard deviation (<1); however, for the HW setup, the deviation is zero. For the failed *lock* scenario, the latency was decreased by a 67.9%. Lastly, RIOT mutexes do not implement priority inheritance, making it unfeasible to reproduce the set of experiments related to this feature.

For the HW configuration of FreeRTOS, the successful and failed *lock* APIs present an improvement of 51.8% and 18.5%, respectively. For the successful and failed *unlock* APIs, there is an improvement of 87.8% and 11.1%, respectively. Notwithstanding, for the priority inheritance related experiments, FreeRTOS presents very high latency for the SW configuration, due to the need of moving multiple threads between several different linked lists. In this experiments, the HW setup reduces the *lock* and *unlock* APIs latency by 95.5% and 76.4% respectively.

For the failed *unlock* API scenario, Zephyr in the SW setup shows a small standard deviation (<1), which, in the HW setup, becomes zero. While in the success *unlock* case, there is an improvement of 550.9%. For both successful and failed *lock* scenarios, the HW configuration decreases the latency by 23.4%. Lastly, for the priority inheritance scenario, the HW setup shows improvements of 461.7% for the *lock* API, and 2615.6% for the *unlock* API. These improvements are a result of Zephyr's software priority inheritance algorithm performing multiple checks to the status of both threads involved, while the HW setup implements all of these checks with combinational logic.

3) Semaphores: Semaphores provide two main services: *give* and *take*. For the sake of completeness, we devised two different experiments to evaluate the different scenarios when both of these operations are requested by a thread. For the first experiment, a singular thread performs a *give* and a *take* sequentially, resulting in a *give* without any waiting, and a *take* with ready data. The second experiment evaluates the remaining conditions for the two APIs. One thread executes a *take* without any data waiting, yielding the execution to another thread that executes a *give* operation, triggering an explicit scheduling point. RIOT HW configuration improves the *give* API performance by 109.7 when a thread is waiting, and by 165.3% when there is no thread waiting. Regarding the *take* operation, the latency is reduced by 67.5% and 54% when there is and there is no data ready, respectively.

FreeRTOS presents the higher latencies for the operations that imply thread state modifications and consequent rescheduling, due to the burden cycles for managing linked lists. Thus, in the HW configuration, the higher latency reduction occurs for the *give* service with thread waiting (decrease in latency of 54.7%) and the *take* service without data ready (decrease of 89.6%). The two other scenarios present an improvement of 24.9% and 11.7% for a *give* without thread waiting and a *take* with data ready, respectively.

Finally, Zephyr's *take* service does not present a significant advantage in the HW configuration, apart from a marginal improvement in the standard deviation. However, the *give* service takes significant advantage from the hardware acceleration, by presenting improvements of 747% and 248.9% for both operations, i.e., with and without thread waiting, respectively.

4) Message Queues: There are two main operations regarding message queues, i.e., *send* and *receive*. Despite both operations may cause threads to be suspended or resumed, the key feature regarding these operations is the message size

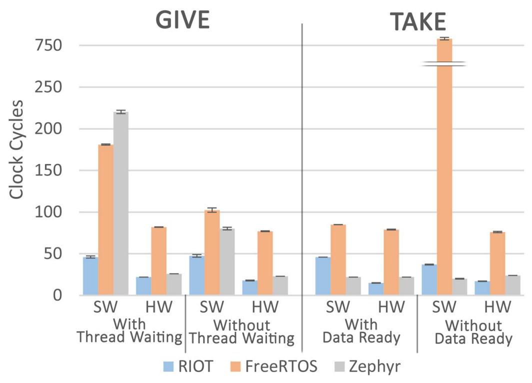

Fig. 7: Semaphore APIs latency.

(related to memory transactions). For that reason, the experiments consist in sending and receiving a message through a message queue, changing and checking its content between each transaction (ensure correctness), as well as increasing the size of the message between each experiment. Results are depicted in Figure 8.

Unlike the other OSes, RIOT does not copy any data to and from memory when performing *send* and *receive* operations. Instead, RIOT gives access to the pointer to the original data to the receiving thread. Naturally, this is a design decision that favours performance while impacting security. Since only the pointer is copied, RIOT presents a constant latency for both *receiving* and *sending* data through message queues, regardless of the message size. Therefore, hardware acceleration only proves beneficial to RIOT, performance-wise, for messages with a small size (lower than 8 words). In turn, FreeRTOS implementation for entering and exiting critical code sections is considerably more complex than in the other RTOSes, which has a noticeable impact, particularly for small messages. Since the critical sections code is used in both SW and HW configurations, the latency of both operations for the HW setup is higher than the other two RTOSes, despite the message size.

Zephyr's memory copy algorithm performs additional control checks. This results in higher latencies for bigger messages. For the HW setup, most of these operations and checks are implemented in hardware, and thus Zephyr is the OS that presents the higher performance improvement in data transactions through message queues, reaching 238.8% in the *send* operation, and 230.7% in the *receive*.

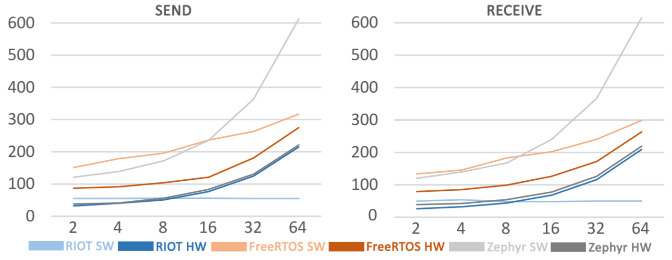

Fig. 8: Message Queues API latency by message size.

TABLE II: Thread-Metric benchmark results.

OS		Basic Processing	Cooperative Scheduling	Preemptive Scheduling	Message Processing	Synchronization
RIOT	SW	67902	4020065	1956595	7103937	7127183
	HW	67902	5868621	2849510	10658732	15068628
(%)		0	45.98	45.64	50.04	111.42
Zephyr	SW	63399	1141517	705823	2828726	6853456
	HW	63400	1249308	717260	7151998	13725909
(%)		0.001	9.44	1.62	152.83	100.28
FreeRTOS	SW	59222	1612707	753302	2362369	3675268
	HW	63165	4846436	2086273	3641817	4451240
(%)		6.66	200.52	176.95	54.16	21.11

C. RTOS performance

To evaluate the overall system's performance, we use the Thread-Metric Benchmark Suite. This synthetic suite implements several benchmarks that stress a singular RTOS-related service. In this work, 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 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 (i.e., a higher loop count indicates better performance). Table II summarizes the results gathered from running each benchmark in periods of 30 seconds. The values presented correspond to the average of 100 samples for a specific benchmark and for both system configurations (SW and HW). An additional row presents the percentage comparison between the HW and SW setups, where positive values correspond to an improvement on the HW over the SW.

The results related to RIOT are in tandem with the results presented in the previous subsection, i.e., the API latency. Since the *Basic Processing* benchmark does not stress any kernel-related operation, there was no change from running with or without hardware acceleration. Both *Cooperative* and *Preemptive Scheduling* present a performance increase of nearly 46%, highlighting that the three main kernel services involved in these experiments (*schedule*, *thread suspend*, and *thread resume*) have a considerable impact on the system performance. The *Message Processing* benchmark presents a performance increase of 50%. Although the hardware implementation still has to copy the same amount of data, most of the memory management is performed in hardware. Lastly, the *Synchronization* benchmark shows the best results, with a performance increase of 111%.

Zephyr presents only a marginal advantage for both *Cooperative* and *Preemptive Scheduling*. This is related to the already optimized scheduling algorithm, already observed in section IV-B1. However, for the *Message Processing* benchmark, there is a significant performance increase of over 152% for the HW configuration. This finding corroborates the results from section IV-B4, where Zephyr was the RTOS that has proven to benefit the most from hardware-acceleration in the message queues processing. The *Synchronization* benchmark also has non-negligible performance increase of 111%.

Among all evaluated RTOSes, FreeRTOS is the only that follows a tick-driven implementation. For this reason, FreeRTOS presents a 6.7% performance increase in the *Basic Processing* benchmark, due to overhead imposed by the *schedule* API. As per the results presented in the previous sections, FreeRTOS APIs related to the scheduling process are the ones that benefit the most from hardware acceleration. This is corroborated in the results of the *Cooperative* and *Preemptive Scheduling* benchmarks, with the HW setup achieving performance increases of 200% and 177%, respectively. Finally, for the remaining two benchmarks, i.e., *Message Processing* and *Synchronization*, the HW setup presents reasonable performance advantages - 54% and 21%, respectively.

D. Hardware Resources

The key feature which has a higher impact on the hardware resources is the number of supported hardware threads. As aforementioned, the data related to each thread is saved by the hardware accelerator in a specific layout which optimizes accessibility on the combinational logic. Additionally, per the information provided in Figure 4, several fields in the node structure can have different sizes. There is a node per hardware thread. Increasing the number of threads or priorities, often results in increasing the number of bits in specific fields. These fields, such as the TID or priority, are propagated throughout the whole accelerator, which naturally, significantly increases the resource consumption. This is illustrated in Figure 9, where we gradually increased (using power of two) the number of threads supported in the hardware. Since we have assigned each thread a unique priority, the number of priorities always matches the number of threads, which also contributes to the resources consumption increase. In this graph, we show only the number of LUTs, Muxes, and Flip-Flops (FFs) consumed by the hardware accelerator (without the resources used by the RISC-V), as the other resources are kept the same.

We have also evaluated the trade-off between the hardware resources consumed by the CHAMELIoT and the number of supported hardware threads, but now including the RISC-V core. Table III summarizes the FPGA resources in terms of LUTs, Muxes, and FFs, for the Rocket core, and the Rocket with CHAMELIoT supporting 2, 4, 8, and 16 threads. For the baseline, we have deployed only the Rocket core on the FPGA with all the OSes implementation in software. When migrating services to hardware, in the smallest setup, i.e., Rocket + CHAMELIoT with 2 threads, the hardware accelerator adds only a small resource increase of 10.5% for the LUTs and nearly 13.3% for the FFs. However, for a typical IoT low-end device configuration, e.g., Rocket + CHAMELIoT with 8 hardware threads, the resource consumption increase is around 35.5% of LUTs, 42,5% of Muxes, and 18.4% of FFs.

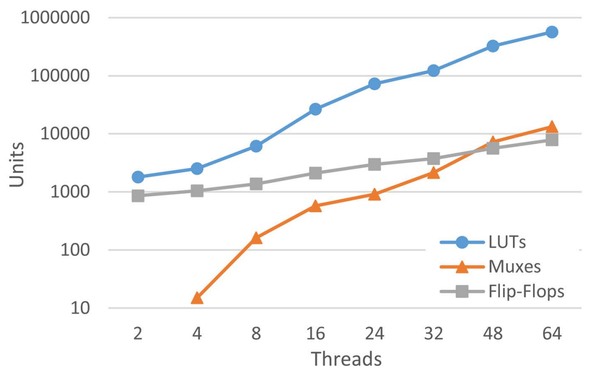

Fig. 9: FPGA resources required by the CHAMELIOT.

TABLE III: FPGA resource consumed by CHAMELIOT integrated with Rocket core.

	LUTs	Muxes	Flip-Flops
Rocket	17246	381	10096
Rocket + CHAMELIOT (2T)	19050 (+10.5%)	374 (-1.8%)	11440 (+13.3%)
Rocket + CHAMELIOT (4T)	19753 (+14.5%)	396 (+3.9%)	11630 (+15.1%)
Rocket + CHAMELIOT (8T)	23361 (+35.5%)	543 (+42.5%)	11950 (+18.4%)
Rocket + CHAMELIOT (16T)	43723 (+153.5%)	959 (+151.7%)	12683 (+25.6%)

V. DISCUSSION

Adaptability vs Performance. We have developed our framework mainly considering RIOT, Zephyr, and FreeRTOS. As this project evolves, we expect to support other OSes, kernel features, and hardware platforms. Consequently, we envision an impact on the configurability points (and complexity) of the accelerator, which, in turn, may lead to some performance degradation. One clear example is that some services from Zephyr only have marginal benefits in the hardware configuration. The trade-off between adaptability and performance has to be made on a per-case basis, as implementing certain features or configurability points may cause tolerable performance degradation in some cases, while in others it would be beneficial to keep these features in software.

Reusability and Adoption. A major goal of CHAMELIOT is reusability and widespread adoption. However, current implementation encompasses a tightly-coupled design, mainly provided as a Rocket core accelerator. This somehow limits the possibility of being adopted by other RISC-V designs and platforms. Therefore, supporting a loosely-coupled configuration will enable the accelerator to be integrated with other systems. Additionally, the Building and Configuration tool is on our mid-term roadmap. The development of this tool will ease the process of configuring and building the whole system, from hardware to software.

Additional Kernel Services. As mentioned in section II-D, 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. Other services may have to be implemented to support features from other OSes, e.g., different scheduling policies. Lastly, some of the current services may have to be modified or have to be enhanced with additional configurability points, to ease the support for additional OSes. For instance, additional or different error messages to specific events.

VI. RELATED WORK

Two distinct approaches can be identified regarding OS acceleration in hardware [25]: (1) *Reconfigurable OS*, which refers to software-based OSes enhanced with capabilities to manage reconfigurable hardware and execute hardware tasks; and (2) *Hardware-accelerated OS*, which are focused on minimizing kernel software execution time by migrating kernel services to hardware. Some of the most prominent reconfigurable OSes are R3TOS [15] and ReconOS [16]. While R3TOS leverages FPGA reconfigurability to provide a reliable and fault-tolerant OS focused on hardware tasks, ReconOS allows the scheduling of both hardware and software threads.

Concerning the hardware-accelerated OSes, they can be classified according to the way they are integrated with other hardware and software components. Hardware integration is closely related with target CPU architecture and how the accelerators are coupled with the core (tightly or loosely approaches). For instance, HThreads [17], HartOS [19], and SEOS [20] are implemented as loosely-coupled accelerators connected via a standard peripheral bus to a Microblaze CPU, which implies the final system to be deployed in an FPGA platform. On their turn, RT-SHADOWS [21] and ARTESSO-TC [26] explore the ubiquitous Arm architecture by modifying its pipeline to support tightly-coupled accelerators. Similarly, ARPA-MT [18] follows the same tightly-coupled approach to a MIPS32 architecture. Having the adoption rate and scalability in mind, ARTESSO-LC [10] implements hardware acceleration in a loosely-coupled fashion, which connects the Arm core through a standard AMBA peripheral bus. The emergence of RISC-V has enabled the creation of hardware accelerators in a tightly-coupled fashion. For instance, OSEK-V [22] implements highly tailored modifications to the RISC-V pipeline to maximize the OS performance.

The software integration of hardware-accelerated OSes is related to the ability of adapting legacy OS applications or developing new ones using hardware acceleration. Hardware-accelerated OSes can achieve this integration by providing their hardware functions as external features that can replace software OSes internals. On the other hand, these OSes can also act as the main OS, requiring current systems to be re-developed and replace the software OS with the new hardware-accelerated OS. Furthermore, the API provided by each of these OSes is directly related with how the OSes can be integrated with the software. The hardware-accelerated OSes API can be unique to the OS, follow a standard like POSIX, or it can be mapped to other API of different OSes. ARPA-MT, HartOS, and OSEK-V all provided their own unique API and are to be used as standalone OS in order to maximize performance. Both ARTESSO-TC and ARTESSO-LC, despite having to be used the acting OS, they provide an API that can

TABLE IV: Comparison of hardware-accelerated OSes.

Hardware RTOS	HThreads	ARPA-MT	HartOS	SEOS	RT-SHADOWS	OSEK-V	ARTESSO-LC	ARTESSO-TC	CHAMELIoT
CPU Architecture	Microblaze	MIPS32	Microblaze	Microblaze	Arm	RISC-V	Arm	Arm	**RISC-V**
Coupling Approach	Loosely	Tightly	Loosely	Loosely	Tightly	Tightly	Loosely	Tightly	**Tightly+Loosely**
Supported OS	Itself	Itself	Itself	Multiple	Multiple	Itself	Itself	Itself	**Multiple**
API	POSIX	Unique	Unique	Mapped	Mapped	Unique	Mapped	Mapped	**Mapped**
Scheduling	✓	✓	✓	✓	✓	✓	✓	✓	✓
Thread Management	✓	✓	✓	✓	✓	✗	✓	✓	✓
Synchronization	✓	✓	✓	✓	✓	✓	✓	✓	✓
Communication	✓	✗	✗	✓	✗	✗	✓	✓	✓

be mapped to fit other Arm cores. Similarly, HThreads provides a POSIX-like API. Lastly, with the goal of maximizing the adoption, both SEOS and RT-SHADOWS can be used to accelerate other COTS software OS through their interface that can be used interchangeably with the software API.

Table IV summarizes all the hardware-accelerated OSes solutions previously discussed, presenting their kernel services currently supported in hardware, API, coupling approach, target CPU architecture, and supported OSes. It also depicts that CHAMELIoT implements all the most important kernel services, while also aiming for greater adaptability and adoption rate by targeting an open-source ISA (RISC-V), being implemented in two different approaches (tightly- and loosely-coupled), and providing a modular API capable of being mapped and support multiple COTS software RTOSes.

VII. Conclusion

In this paper, we presented CHAMELIoT, an agnostic hardware OS framework for FPGA-based IoT devices leveraging RISC-V. We have deployed and evaluated our system with three different OSes: RIOT, FreeRTOS, and Zephyr. Results demonstrated that latency can be decreased up to 96.3%, and for most services, the jitter was removed. At the application level, we observed performance increases of up to 200%.

Acknowledgments

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.

References

[1] H. Sundmaeker, P. Guillemin, P. Friess, and S. Woelfflé, “Vision and Challenges for Realizing the Internet of Things,” *Cluster of European Research Projects on the Internet of Things, EU Commision*, 2010.

[2] C. Perera, C. H. Liu, and M. Chen, “A Survey on Internet of Things From Industrial Market Perspective,” *IEEE Access*, 2014.

[3] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, “A Survey on the Edge Computing for the Internet of Things,” *IEEE Access*, 2018.

[4] D. Oliveira, M. Costa, S. Pinto, and T. Gomes, “The Future of Low-End Motes in the Internet of Things: A Prospective Paper,” *MDPI Electronics*, 2020.

[5] M. Valdés, J. Rodriguez-Andina, and M. Manic, “The Internet of Things: The Role of Reconfigurable Platforms,” *IEEE Industrial Electronics Magazine*, 2017.

[6] M. Elnawawy, A. Farhan, A. Nabulsi, A. Al-Ali, and A. Sagahyroon, “Role of fpga in internet of things applications,” in *IEEE Int. Symp. on Signal Proc. and Inf. Tech.(ISSPIT)*, 2019.

[8] A. Waterman, “Design of the RISC-V Instruction Set Architecture,” Ph.D. dissertation, UC Berkeley, 2016.

[7] K. Asanovic and D. Patterson, “Instruction Sets Should Be Free: The Case For RISC-V,” EECS Department, University of California, Berkeley, Tech. Rep., 2014.

[9] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, P. Dabbelt, J. Hauser, A. M. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moretó, A. Ou, D. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, “The Rocket Chip Generator,” 2016.

[10] N. Maruyama, T. Ishikawa, S. Honda, H. Takada, and K. Suzuki, “ARM-based SoC with Loosely Coupled type Hardware RTOS for Industrial Network Systems,” 2014.

[11] M. Silva, D. Cerdeira, S. Pinto, and T. Gomes, “Operating Systems for Internet of Things Low-End Devices: Analysis and Benchmarking,” *IEEE Internet Things J.*, 2019.

[12] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, “Operating Systems for Low-End Devices in the Internet of Things: A Survey,” *IEEE Internet Things J.*, 2016.

[13] S. R. Corp., “RISC-V Market Analysis The New Kid on the Block,” 2019.

[14] P. D. Schiavone, F. Conti, D. Rossi, M. Gautschi, A. Pullini, E. Flamand, and L. Benini, “Slow and Steady Wins the Race? A Comparison of Ultra-Low-Power RISC-V cores for Internet-of-Things Applications,” in *2017 27th Int. Symp. on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2017.

[15] X. Iturbe, K. Benkrid, C. Hong, A. Ebrahim, R. Torrego, and T. Arslan, “Microkernel Architecture and Hardware Abstraction Layer of a Reliable Reconfigurable Real-Time Operating System (R3TOS),” *ACM Trans. Reconfigurable Technol. Syst.*, 2015.

[16] E. Lübbers and M. Platzner, “ReconOS: Multithreaded Programming for Reconfigurable Computers,” *ACM Trans. on Embedded Computing Systems*, 2009.

[17] J. Agron, W. Peck, E. Anderson, D. Andrews, E. Komp, R. Sass, F. Baijot, and J. Stevens, “Run-Time Services for Hybrid CPU/FPGA Systems on Chip,” in *27th IEEE Int.l Real-Time Systems Symp. (RTSS’06)*, 2006.

[18] A. Oliveira, L. Almeida, and A. Ferrari, “The ARPA-MT Embedded SMT Processor and Its RTOS Hardware Accelerator,” *IEEE Trans. Ind. Electron.*, 2011.

[19] A. Lange, K. Andersen, U. Schultz, and A. Sørensen, “HartOS - A hardware implemented RTOS for hard real-time applications,” 2012.

[20] S. E. Ong, S. C. Lee, N. B. Z. Ali, and F. A. B. Hussin, “SEOS: Hardware Implementation of Real-Time Operating System for Adaptability,” in *First Int. Symp. on Computing and Networking*, 2013.

[21] T. Gomes, P. Garcia, S. Pinto, J. Monteiro, and A. Tavares, “Bringing Hardware Multithreading to the Real-Time Domain,” *IEEE Embedded Syst. Lett.*, 2016.

[22] C. Dietrich and D. Lohmann, “OSEK-V: Application-Specific RTOS Instantiation in Hardware,” in *Proc. of the 18th ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools for Embedded Systems*, 2017.

[23] W. Hofer, D. Lohmann, F. Scheler, and W. Schröder-Preikschat, “SLOTH: Threads as Interrupts,” in *2009 30th IEEE Real-Time Systems Symp.*, 2009.

[24] T. Gomes, P. Garcia, F. Salgado, J. Monteiro, M. Ekpanyapong, and A. Tavares, “Task-Aware Interrupt Controller: Priority Space Unification in Real-Time Systems,” *IEEE Embedded Syst. Lett*, 2015.

[25] G. J. Brebner, “A Virtual Hardware Operating System for the Xilinx XC6200,” in *Proc. of the 6th Int. Workshop on Field-Programmable Logic, Smart Applications, New Paradigms and Compilers*, 1996.

[26] N. Maruyama, T. Ishihara, and H. Yasuura, “An RTOS in Hardware for Energy Efficient Software-based TCP/IP processing,” in *IEEE 8th Symp. on Application Specific Processors (SASP)*, 2010.