Leveraging RISC-V to build an open-source (hardware) OS framework for reconfigurable IoT devices

Miguel Silva
Centro ALGORITMI, Universidade do Minho
Guimarães, Portugal
miguel.silva@dei.uminho.pt

Tiago Gomes
Centro ALGORITMI, Universidade do Minho
Guimarães, Portugal
mr.gomes@dei.uminho.pt

Sandro Pinto
Centro ALGORITMI, Universidade do Minho
Guimarães, Portugal
sandro.pinto@dei.uminho.pt

ABSTRACT

With the growing interest in RISC-V systems and the endless possibilities of creating customized hardware architectures, we introduce the first proof of concept (PoC) implementation of ChamelIoT, the first open-source agnostic hardware operating system (OS) framework for reconfigurable Internet of Things (IoT) low-end devices. At this stage, ChamelIoT, leveraging the Rocket Custom Co-Processor Interface (RoCC), provides hardware acceleration support for thread management and scheduling of three different OSes: RIOT, Zephyr, and FreeRTOS. This paper overviews the overall ChamelIoT architecture and describes the implementation details of the current PoC deployment. Our first experiments were carried out on a Xilinx Arty-35T FPGA Evaluation kit and the preliminary results are very promising, showing that the desired agnosticism and flexibility can be achieved with determinism and performance advantages at a reasonable cost of hardware resources.

KEYWORDS

Hardware Acceleration, Operating Systems, RISC-V, IoT.

ACM Reference Format:
Miguel Silva, Tiago Gomes, and Sandro Pinto. 2021. Leveraging RISC-V to build an open-source (hardware) OS framework for reconfigurable IoT devices. In *Proceedings of Fifth Workshop on Computer Architecture Research with RISC-V (CARRV 2021).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Internet of Things (IoT) is revolutionizing the Internet of the future by connecting countless smart devices over a massive and collaborative network infrastructure. With the ever-growing interest in this topic, the variety of use cases, constraints, and requirements is considerably increasing [20, 23]. To cope with such diversity, and due to the lack of turn-key solutions for low-end IoT devices, the software and hardware development processes face several trade-offs regarding performance, power consumption, form-factor, just to name a few. Such trade-offs are often dictated by the software application, usually an embedded operating system (OS), whose architecture and design decisions (e.g., kernel and scheduling mechanisms) have a big impact on the overall system behavior [14, 22].

The migration of OS kernel services to hardware is not a new endeavor since it provides several advantages in terms of performance, predictability, and determinism [1, 2, 12, 15, 16, 18]. These services can be implemented following two well-established approaches: (i) tightly-coupled, where the accelerator is embedded with the main core datapath [9, 12]; or (ii) loosely-coupled, which implies the accelerator to be implemented outside the core, connected by standard communication buses [11, 18]. Despite each approach providing several advantages, they fell short in getting traction in the industry due to the several issues steamed by proprietary ISAs and closed-source architectures, e.g., Arm [18].

RISC-V is a novel open-source instruction set architecture (ISA) that follows a reduced instruction set computer (RISC) design [4, 26]. It was created to be very modular and customizable, allowing it to scale from tiny microcontrollers to powerful processors. Moreover, RISC-V is gaining more attention from the industry by enabling the development of specialized solutions that attended to the heterogeneity of requirements and constraints across different applications and use cases [6, 7, 10, 24]. The open-source model of RISC-V creates new opportunities to explore innovative approaches of accelerating OS services in hardware, which can be easily integrated with any standard RISC-V core.

ChamelIoT is a framework for reconfigurable IoT platforms that aims at building an agnostic hardware OS framework. By leveraging the RISC-V architecture, ChamelIoT plans to solve challenges that once hampered the adoption of hardware-accelerated OSes, while providing increasing determinism, performance, and real-time guarantees. The ChamelIoT architecture is composed of hardware accelerators that can be instantiated in a tightly- or loosely-coupled configuration. These accelerators are designed to support OS services such as the scheduling process, thread management, time control, and synchronization mechanisms.

This paper details the first proof-of-concept (PoC) implementation of the ChamelIoT framework. Currently, it is being deployed on a tightly-coupled approach, resorting to the Rocket Custom Co-Processor Interface (RoCC) to instantiate the co-processor in hardware, accessible through custom instructions. At this stage, ChamelIoT encompasses the acceleration of the scheduling process and part of the thread management of three different IoT OSes: RIOT, Zephyr, and FreeRTOS. The preliminary results are quite promising, showing that the desired agnosticism and flexibility can be achieved with determinism and performance advantages at a reasonable cost of hardware resources.

2 GOALS

Reconfigurable technology, namely Field Programmable Gate Arrays (FPGA), is gaining special attention in the embedded IoT arena thus providing augmented capabilities for implementing reprogrammable and customized hardware accelerators for a wide range of applications, e.g., machine learning, networking, security

CARRV 2021, Date, Location
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

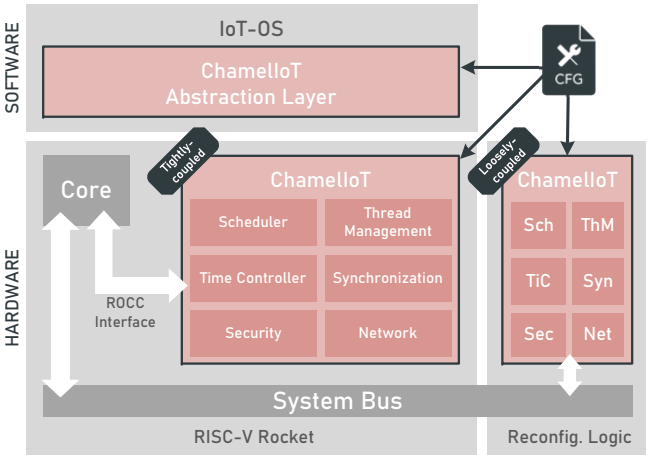


Figure 1: Standard hardware-software stack with ChamelIoT accelerator.

[8, 13, 19, 25]. Embedded FPGAs such as the ones provided by QuickLogic or low-power FPGAs from the Lattice portfolio are seeing increasing applicability in low-end IoT devices due to the possibility of developing, in a single semiconductor, multiple adjacent market segments, while constraining the size, weight, power, and cost (SWaP-C). ChamelIoT vision is aligned with such platforms and aims at providing:

Real-Time and Determinism. ChamelIoT shall be able to provide hard real-time guarantees and bounded worst-case execution time. Predictability shall not be dependent on the number and priorities of threads/tasks existing in the system.

Performance. ChamelIoT shall be able to execute accelerated OS services in a smaller amount of time than the ones provided by standard IoT OSes.

Agnosticism. ChamelIoT shall be able to run unmodified applications from different IoT OSes while providing a software abstraction layer (i.e., ChamelIoT wrappers / API) for creating retro-compatibility among them.

Flexibility. ChamelIoT shall be able to provide a widely configurable architecture and an easy-to-use tool so that it can be (i) adapted to a wide variety of architectures and platforms and (ii) tuned to fit specific application's requirements.

Power Consumption. ChamelIoT shall present a reduced or equivalent power consumption than standard IoT OSes. Accelerated OS services are expected to create additional idle times, which shall be used to explore advanced low-power modes.

3 ARCHITECTURE

Figure 1 depicts a standard hardware-software stack with the RISC-V processor and the ChamelIoT accelerators. The RISC-V processor is based on Rocket [3], which is a 64-bit implementation that integrates both L1 and L2 data caches and provides the RoCC interface [17]. The RoCC interface was specially designed to help in attaching accelerators to Rocket, accessible through a standard *R-type* instruction format. The main goal of ChamelIoT is to provide hardware support to OS services and libraries, such as the scheduling process, thread management, synchronization, networking, and security. This custom hardware-software architecture is managed from an external tool that allows the configuration of the hardware components, such as the RISC-V core and the ChamelIoT accelerator, and the software layer that is responsible to interface different implementations of OS components with ChamelIoT services.

The current PoC supports the ChamelIoT co-processor connected through the RoCC interface, accessed through custom instructions added to the RISC-V ISA. At this stage, ChamelIoT supports priority-based scheduling algorithms and multiple thread management with ready-queues. To provide these hardware-based services to an OS, we modified the OS's internals, by remapping software-based components to agnostic ChamelIoT functions.

3.1 Operating Systems for the IoT

Among the list of available OSes that can best serve low-end IoT devices [14, 22], we selected RIOT, Zephyr, and FreeRTOS, due to (i) their broad popularity and applicability in IoT applications and (ii) the continuous support from their respective open-source communities. Moreover, such OSes provide enough variability with regard to the main design points we consider important to evaluate and provide support with ChamelIoT.

RIOT implements a microkernel-like architecture resorting to a tickless preemptive scheduler based on descending priorities [5]. To fulfill the real-time requirements, RIOT also guarantees the execution of kernel tasks and inter-process communication (IPC) with limited interrupt latency (around 50 clock cycles) and provides low overhead multi-threading support. Designed with the IoT ecosystem in mind, its main features are real-time capabilities, support to low-power wireless devices, and a built-in network stack compliant with many standards and protocols.

Zephyr implements a microkernel-based architecture with a tickless scheduler and descending priorities scheme. The scheduler's ready queue can be further configured as simple linked-list, red/black tree, and traditional multi-queue. These bring several configurations that result in different trade-offs between binary size, memory footprint, performance, and determinism. Zephyr's state mechanism is based on a set of seven flags that are assigned in the scheduling process and each thread can have multiple active flags, allowing for multiple states at the same time. Zephyr also provides a built-in network stack for connectivity.

FreeRTOS is one of the most widely available OSes due to its portability, open-source community, and focus on providing real-time capabilities. FreeRTOS follows a preemptive tick-driven scheduling policy that dictates the thread scheduling according to their assigned (ascending) priorities. This OS follows a simple state machine where each thread is assigned with one of four unique states, two of which imply the thread is ready to be executed. Network stack support is provided as software libraries and upstreamed as a single package in the Amazon FreeRTOS.

Table 1 summarizes key design points of the internals of each OS. These three OSes offer a good degree of variability, especially considering their scheduling algorithms and thread management, which is the focus of the current PoC. While Zephyr offers different options to implement the ready queue mechanism, to the extent of this PoC we just support the default multi-queue, where each

Table 1: Key aspects of each OS regarding their scheduling policy implementation.

Key aspects	RIOT	Zephyr	FreeRTOS
Ready Queue	Multi-queue	Multi-queue	Multi-queue
States	14	8	4
Scheduling Points	Tickless	Tickless	Tick-based
Priority	Descending	Descending	Ascending

priority level has its own linked list. FreeRTOS is the single OS with a tick-driven scheduler, using ascending priorities.

3.2 RISCV-V Core

ChamelIoT is currently deployed on the open-source SiFive E300 platform featuring an E31 RISC-V core (RV32-IMAC), which supports atomic (A) and compressed (C) instructions, for higher performance and better code density, respectively. This core is implemented with the Rocket chip generator and provides a single-issue in-order 32-bit pipeline, with a peak sustained execution rate of one instruction per clock cycle [21]. Additionally, it provides a platform-level interrupt controller (PLIC), a debug unit, and a wide variety of peripherals. Lastly, the E300 platform contains two TileLink interconnection interfaces, one of which can be used to attach custom accelerators. ChamelIoT leverages the RoCC interface to provide specialized hardware OS services tightly-coupled to the E31 core.

The RoCC interface is further divided into sub-interfaces that allow for communication between co-processors and the core while providing direct access to the first level of the data cache. To communicate with the core, RoCC defines an extension to the RISC-V ISA by introducing a custom instruction that follows the *R-type* format, depicted in Figure 2. The bit fields on this instruction specify the target co-processor and operation. The *opcode* field can only be assigned with one of four predefined values that are allocated to each co-processor. The fields *rd, rs1, and rs2* specify the destination (rd) and source (rs1 and rs2) registers, used to transfer data with the co-processor. Respectively, the *xd, xs1, and xs2* bits are set to identify which of corresponding registers, *rd, rs1, and rs2*, are being used. The last field, *funct7*, is a set of bits that, together with the software, are used to indicate what function the co-processor has to execute, acting as a specific opcode for each co-processor.

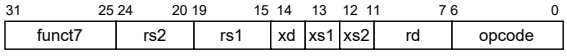


Figure 2: RoCC instruction format.

4 PROOF-OF-CONCEPT IMPLEMENTATION

At this stage of development, ChamelIoT is focusing on proving support to the scheduling service of the selected OSes. Since it is currently deployed as a co-processor through the RoCC interface in a 32-bit Rocket core, it presents some limitations on the amount of data transferred on an instruction-basis scheme. Therefore, and as depicted in Figure 2, the accelerator's input data width is only two 32-bit words, while the output data is a single 32-bit word. Moreover, the amount of data related to each thread managed by ChamelIoT

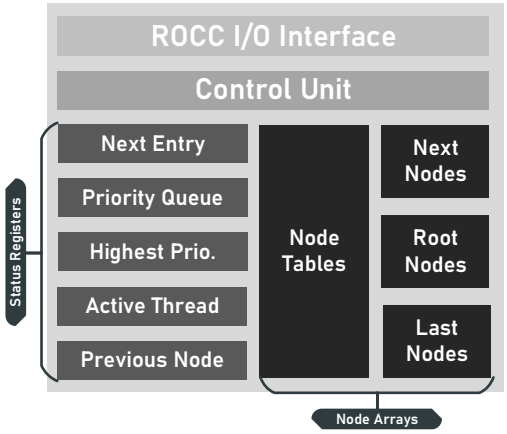


Figure 3: ChamelIoT hardware architecture.

also has to be limited in order to save hardware resources. For this reason, it is still to be understood the trade-offs of providing full support for each Thread Control Block (TCB) in hardware. Thus, at the moment, each OS is still responsible for managing its TCB implementation. Nonetheless, since the thread state and priority play a major role in the scheduling process, these elements are currently managed at the hardware.

ChamelIoT provides support for the multiple ready-queues by implementing a linked list per priority level. Thus, an hardware infrastructure is required to properly add or remove each thread from the correct linked list when needed, without incurring additional performance penalties to the system. This often implies resource allocation to save information about each list and other relevant information. Lastly, ChamelIoT provides a control unit to manage the communications interface and data translation coming from the CPU. At the moment, ChamelIoT expects the software to fully manage the operation of adding and removing threads and the timing of each scheduling point.

4.0.1 *Hardware Architecture.*

The co-processor architecture can be divided into three main blocks: (1) the *Control Unit*; (2) *Status Registers*; and (3) *Node Arrays*, as illustrated by Figure 3. The *Control Unit* is responsible for managing the interaction with the RoCC I/O interface, parsing the data needed on each instruction, decoding the *funct7* field, and controlling the remaining blocks to execute the required function. Each element from the *Status Registers* block contains information that needs to be accessible at the beginning of each operation to avoid introducing any delays. Thus, whenever an operation is issued to ChamelIoT each element of the *Status Registers* has to be updated in the following way:

- The ***Next Entry*** register points to the next available position in the thread node table;
- ***Priority Queue*** saves the ID of each non-empty ready-queue;
- The ***Highest Priority*** register holds the priority value of the next thread to be executed, according to the OS priority scheme (ascending or descending);

- ***Active Thread*** holds the Thread ID (TID) of the currently running thread;
- ***Previous Node*** register has the TID of the previous thread on the linked-list.

The *Node Arrays* block implements arrays of registers according to either the maximum number of defined threads and priorities. The *Node Table* is the central component of ChamelIoT since it is where all the information related to each thread is stored and managed. Each thread present in the system has a corresponding node in this table with the information shown in Figure 4. The *data* field (32-bit width) is used to save the TCB pointer (some OSes, e.g. FreeRTOS, use a pointer instead of a TID), while the remaining fields are mostly used in the scheduling and list management tasks: the *dirty* bit is used to indicate whether or not that table position is being used; the *next* field holds a TID pointer to the next thread on the specific linked-list; and both the *priority* and *state* fields are updated by the OS and used by ChamelIoT to determine which task is going to be executed next and if the thread should be in the ready-queue.

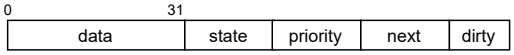


Figure 4: Thread node structure.

The current implementation of ChamelIoT replicates a system where a linked list is created for each priority defined in the system. These lists together form the ready-queue that sets the next active thread. By default, whenever a new thread is added, it is not automatically added to the ready-queue, being added to a given linked list only when its state is changed by the OS flagging the thread is ready to run. The remaining components of *Node Arrays* block are used to manage and iterate each list, containing a set of registers, one per priority level, that hold the following information: (i) *Next Nodes* has the TIDs for the next thread to be executed in each priority level; (ii) the *Root Nodes* have the TID of the first node on each list; and (iii) the *Last Nodes* contains the last nodes.

With the purpose of executing any instruction provided by software without incurring any delay, it is necessary all the information about all nodes associated with each list. This implementation results in a trade-off between hardware resources, performance, and determinism. Alternatively, an implementation similar to a software linked-list, where each operative is done iteratively, would result in fewer hardware resources at the cost of performance and determinism. ChamelIoT schedules the next running thread by selecting, on each priority list, the threads on the ready-queue. Then, according to the implemented priority scheme (ascending or descending), it checks which is the next ready node with that priority level and returns it to the OS, updating its status flags, accordingly. Unless explicitly demanded by software, once a thread is added to a queue, it will remain there until its state is updated.

4.0.2 *Hardware configurability.*

One of the main concerns when deploying a co-processor in FPGA is the amount of hardware resources that are often required to accommodate all the necessary logic. With ChamelIoT, such resources are highly dictated by the hardware configurations required by the final solution. For instance, the number of priorities is directly related to the number of linked-list components that have to be maintained by ChamelIoT in hardware, and consequently, dictates the number of registers required to store data. Likewise, the number of threads has an impact on the number of nodes needed to be allocated in the *Node table*. Moreover, the number of bits required by TID-related fields also increases with the number of threads. Thus, by supporting a customizable number of threads and priorities, ChamelIoT aims at providing fine-grain customization per the application and use-case requirements. The current configurations available for ChamelIoT are number of supporting threads and priorities, number of states, ready-queue states, thread default states, and default priority order.

Table 2: Software functions to interact with ChamelIoT.

Function	*funct7*	*rs1*	*rs2*	*rd*	Description
Add	0	priority	TCB	TID	Adds a thread to the table
Remove	1	TID	none	none	Removes a thread from the table
Active PID	2	none	none	TID	Returns the TID of the running thread
Get TCB	3	TID	none	TCB	Returns the thread TCB
Schedule	4	none	none	TID	Schedules and sets the state of the next thread; changes the state of the current active thread
Set State	5	TCB	state	none	Changes the thread state (might cause ChamelIoT to move the thread to/from the ready queue)
Get PID	6	TCB	none	TID	Returns the thread PID

4.0.3 *Software Interface.*

The OS interacts with ChamelIoT through a custom instruction added to the RISC-V ISA, which allows to share data through *rs1*, *rs2*, and *rd* registers regarding the function specified in the *funct7* register. Thus, it was required to develop a small set of APIs to provide an agnostic adaption layer for each OS being able to use ChamelIoT. Table 2 summarizes the functions currently implemented and accessible by software, as well as the required fields to build the custom instruction to communicate with the RoCC interface. The *Add* and *Remove* functions provide basic thread management functionalities used by the OS during the thread creation/deletion process. To add a thread to the memory table, ChamelIoT requires its priority number and TCB pointer. To remove a thread, ChamelIoT only needs the TID, which basically results in clearing the *dirty* bit associated with that thread.

One common feature present on each OS is the TID to TCB translation, and vice-versa. For instance, RIOT uses the TID to identify and manage threads, while Zephyr and FreeRTOS use TCB data pointers. Therefore, ChamelIoT must also support a mechanism to interchange between a TID and a TCB, which is done via the functions *Get PID* and *Get TCB*. Additionally, through the *Active PID* function, the OS can request to ChamelIoT the TID of the thread that is currently running. Regarding the *Set State* function, it allows the OS to add or remove a thread from the ready-queue. Whenever this action is requested to ChamelIoT, the accelerator checks both thread's previous and next states to evaluate if the thread should be removed or added to the ready queue. Lastly, the function *Schedule*

Table 3: RISC-V core and ChamelIoT (configured with 8 threads and 8 priorities) hardware resources.

Resource	Rocket	Rocket + RoCC	Rocket + ChamelIoT
LUTs	17246	17791 (+3.16%)	21769 (+26.23%)
MMCM	1	1 (+0%)	1 (+0%)
Muxes	381	403 (+5.77%)	485 (+27.3%)
RAM	197	229 (+16.24%)	240 (+21.83%)
SRL	89	89 (+0%)	89 (+0%)
FFs	10096	10362 (+2.63%)	10827 (+7.24%)
IOBUF	58	58 (+0%)	58 (+0%)

must be used by each OS at their scheduling points to request from the co-processor the TID of the thread that must be executed next.

5 EVALUATION AND PRELIMINARY RESULTS

We deployed and evaluated ChamelIoT PoC implementation on a SiFive Freedom E300 running at 65MHz in a Xilinx Arty-35T FPGA board. ChamelIoT was tested with RIOT, Zephyr and FreeRTOS, configured to support a total maximum of 8 threads and 8 priorities. We focus our preliminary evaluation on hardware resources (Section 4.1) and determinism and performance (Section 4.2).

5.1 Hardware Resources

Table 3 details the FPGA resources (obtained from Vivado 2020.2) required to deploy the Rocket RISC-V core (baseline) with and without ChamelIoT, configured to support 8 threads and 8 priorities. Furthermore, we have also evaluated the cost of adding the RoCC interface, which is a major requirement for adding a tightly-coupled co-processor. Adding the RoCC interface has a small impact in the overall system, with the major cost on RAM, which represents an increase of around 16.24% when compared with the Rocket core without the RoCC interface. For the other resources, the impact is negligible, e.g., 5.77% Muxes, 3.16% Look-up Tables (LUTs), 2.63% Flip-Flops (FFs), and 0% for the remaining FPGA components.

When ChamelIoT is added to the system (RoCC interface is also included), the number of LUTs, Muxes, and required RAM increases to 26.23%, 27.3%, and 21.83%, respectively. This is due to the fact that the current implementation of ChamelIoT strongly relies on combinational logic components to satisfy the expected performance and deterministic requirements. A possible approach to minimize the hardware resources could encompass the exploration of a mixed approach with both combinational and sequential logic, shifting the load from LUTs and Muxes to Flip-Flops. Nonetheless, the amount of resources used by ChamelIoT is just a small portion of the whole system when taken the Rocket core also into consideration.

For the purpose of understanding the impact of the number of threads and priorities supported by ChamelIoT in terms of hardware costs, we have evaluated the co-processor in different configurations, varying the number of threads and the number of priorities. Figure 5 depicts the hardware resources required by ChamelIoT with a different number of supported threads and priorities. Since the most hardware-consuming components are LUTs, Muxes, and FFs, the other resources were left aside from this comparison.

From Figure 5a, which depicts the hardware consumption when the system supports a fixed number of priorities but varying the number of threads (up to 32), it is possible to conclude that increasing the number of threads impacts the FPGA resources. Considering that the number of bits required to identify each thread is incremented when the maximum number of threads equals a power of 2, it is expected to observe an increase in the hardware resources. This is mostly reflected in the number of LUTs and Muxes, as the current PoC is mainly implemented with combinational logic.

To some extent, the same phenomenon can be observed in Figure 5b. When the number of supported threads is fixed at 8 and the amount of priorities is increasing, it is possible to identify a nearly linear trend for the number of LUTs and Muxes, while the number of Flip-Flops presents just a small increase.

Despite the bit count of the priority level being incremented at certain points, it does not impact the system in the same way as the thread's bit count, since the priority does not affect the number of nodes required to be allocated in the *Node Table*. In sum, it is clear that the number of threads causes more impact in terms of resources than the number of priorities.

5.2 Determinism and Performance

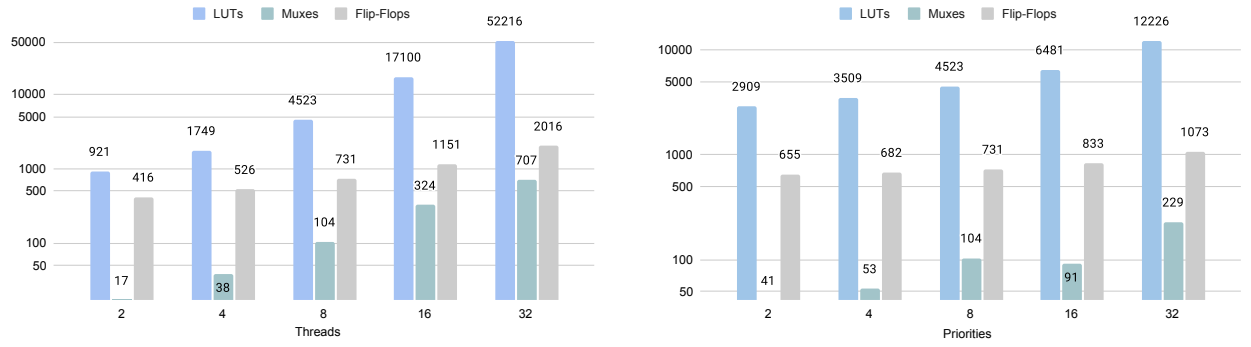
To assess determinism and performance, we have measured the number of clock cycles required to complete the thread selection algorithm, both for the software and hardware implementations. This experiment was repeated 10000 times using the performance counter *MCYCLE* available in the RISC-V core, and the gathered results are summarized in Figure 6.

For the software-based scheduler, Zephyr provides the best results, followed by RIOT, and FreeRTOS. While Zephyr requires, on average, 16 clock cycles, RIOT needs 44 clock cycles and FreeRTOS 403. When the scheduling tasks are performed by ChamelIoT, the required clock cycles are highly reduced: Zephyr and FreeROTS require 8 clock cycles, while RIOT uses only 4 clock cycles. FreeRTOS is the OS that more benefits from ChamelIoT, presenting a reduction of nearly 96%. Regarding the different clock cycles between RIOT and the others, RIOT benefits from the function *Scheduling*, present in Table 2, which returns the TID of the next thread, being this behavior identical to the RIOT's kernel. For both Zephyr and FreeRTOS, it is necessary to convert the TID into the correct TCB data pointer. This requires to use an extra instruction to interact with the RoCC, which results in the double of clock cycles.

For the current experiment, it is clear that FreeRTOS and RIOT are improved by ChamelIoT. However, Zephyr already has a very optimized scheduling process. Nonetheless, from previous research [22], we identified that Zephyr's performance in functionalities like synchronization is considerably worse than the other OSes. Thus, we expect ChamelIoT to present advantages for Zephyr in other dimensions than the ones currently presented in this preliminary evaluation. In the near future, we plan to undergo an extensive evaluation and demonstrate the increasing advantages of ChamelIoT for performance and determinism.

6 DEVELOPMENT ROADMAP

Despite being currently under development, ChamelIoT aims at building a complete and mature framework where several OS services can be migrated and supported in a hardware, using a tightly-



(a) Eight fixed priorities varying the number of threads.

(b) Eight fixed threads varying the number of priorities.

Figure 5: Resources consumed by ChamelIoT with different number of supported threads and priorities.

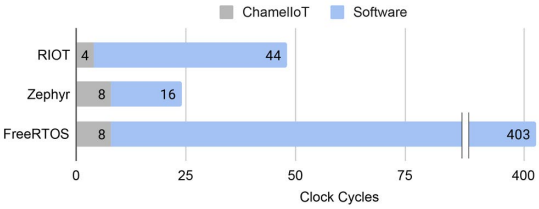


Figure 6: Clock cycles required by the scheduling algorithm of each OS, with and without ChamelIoT.

and/or loosely-coupled fashion. Hereafter, we plan to follow the development in both the hardware and software components, which includes the following features: (i) thread management, (ii) timing control, (iii) synchronization and inter-process communication, (iv) broader OS support, and (v) a configuration tool.

6.1 Hardware

Thread Management: Currently, only a subset of the desired thread management infrastructure is being implemented, which supports the control of the priority and state of each thread. Nonetheless, we aim at extending this support to more TCB data fields, which will enable the implementation of extra features like stack validation or even result in a minimized context switching time.

Timing Control: This improvement will encompass the addition of a hardware timer fully controlled by ChamelIoT, which will allow the migration of the timer service functionalities of each OS to the co-processor. This is expected to improve determinism, in particular at each scheduling point.

Synchronization and IPC: In the near future, ChamelIoT is expected to provide hardware support for synchronization mechanisms like mutexes, semaphores, as well ass IPC features.

Loosely-coupled Accelerator: For reconfigurable platforms enhanced with hard-cores and programmable logic (i.e., FPGA), we plan to implement ChamelIoT as a memory-mapped peripheral. While this approach may slightly hurt predictability, determinism, and performance, when compared to a tightly-coupled configuration, it will enable the deployment of ChamelIoT on a wider number of platforms and architectures.

6.2 Software

OS integration: We aim at providing an easy-to-use and thoroughly documented API that can be further expanded into any OS by software developers. Additionally, we plan to extend the native support of ChamelIoT to other IoT OSes.

Configuration tool: The biggest goal of this work is the realization of an open-source agnostic framework for reconfigurable IoT end-devices, supported by the RISC-V processor architecture. All configuration steps (both for hardware and software modules), OS integration, as well as the final board deployment, should be supported by graphical and user-friendly tools, which will minimize the efforts to interact and use ChamelIoT on a wide number of applications.

7 CONCLUSIONS

In this paper, we present ChamelIoT, an open-source agnostic hardware OS framework for reconfigurable IoT devices. By leveraging the RoCC interface available in a Rocket RISC-V core, we have designed and implemented ChamelIoT as a tightly-coupled co-processor. At this stage, ChamelIoT supports the acceleration of the scheduling process and part of the thread management of RIOT, Zephyr, and FreeRTOS. The given results have shown that our framework is able to bring determinism and performance enhancements to the scheduling process at a reasonable hardware cost. Following the development of ChamelIoT, we will provide support to a broader number of features and expand such functionalities to other kernel services and OSes.

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] Andreas Agne, Markus Happe, Ariane Keller, Enno Lübbers, Bernhard Plattner, Marco Platzner, and Christian Plessl. 2014. ReconOS: An Operating System Approach for Reconfigurable Computing. *Micro, IEEE* 34 (01 2014), 60–71.

[2] Jason Agron, Wesley Peck, Erik Anderson, David Andrews, Ed Komp, Ron Sass, Fabrice Baijot, and Jim Stevens. 2006. Run-Time Services for Hybrid CPU/FPGA Systems on Chip. In *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. 3–12.

[3] K. Asanović, Rimas Avizienis, J. Bachrach, S. Beamer, David Biancolin, Christopher Celio, Henry Cook, Palmer Dabbelt, J. Hauser, Adam M. Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, Jack Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moretó, Albert J. Ou, D. Patterson, B. Richards, C. Schmidt, S. Twigg, Huy Vo, and Andrew Waterman. 2016. The Rocket Chip Generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17.*

[4] Krste Asanovic and David A. Patterson. 2014. *Instruction Sets Should Be Free: The Case For RISC-V.* Technical Report. EECS Department, University of California, Berkeley.

[5] E. Baccelli, C. Gündoğan, O. Hahm, P. Kietzmann, M. S. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch. 2018. RIOT: An Open Source Operating System for Low-End Embedded Devices in the IoT. *IEEE IoT Journal* 5, 6 (Dec 2018), 4428–4440.

[6] SEMICO Research Corporation. 2019. RISC-V Market Analysis The New Kid on the Block. (2019).

[7] Pasquale Davide Schiavone, Francesco Conti, Davide Rossi, Michael Gautschi, Antonio Pullini, Eric Flamand, and Luca Benini. 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)*. 1–8.

[8] Christian Dietrich and Daniel Lohmann. 2017. OSEK-V: Application-Specific RTOS Instantiation in Hardware. In *Proceedings of the 18th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (Barcelona, Spain) *(LCTES 2017)*. ACM, New York, NY, USA, 111–120.

[9] Tim Fritzmann, Georg Sigl, and Johanna Sepúlveda. 2020. Risq-v: Tightly coupled risc-v accelerators for post-quantum cryptography. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2020), 239–280.

[10] Angelo Garofalo, Giuseppe Tagliavini, Francesco Conti, Luca Benini, and Davide Rossi. 2021. XpulpNN: Enabling Energy Efficient and Flexible Inference of Quantized Neural Networks on RISC-V based IoT End Nodes. *IEEE Transactions on Emerging Topics in Computing* (2021), 1–1.

[11] Davide Giri, Kuan-Lin Chiu, Guy Eichler, Paolo Mantovani, N Chandramoorth, and Luca P Carloni. 2020. Ariane+ NVDLA: seamless third-party IP integration with ESP. In *Workshop on Computer Architecture Research with RISC-V (CARRV)*.

[12] Tiago Gomes, Paulo Garcia, Sandro Pinto, João Monteiro, and Adriano Tavares. 2016. Bringing Hardware Multithreading to the Real-Time Domain. *IEEE Embedded Systems Letters* 8, 1 (2016), 2–5.

[13] Tiago Gomes, Filipe Salgado, Adriano Tavares, and Jorge Cabral. 2017. CUTE Mote, A Customizable and Trustable End-Device for the Internet of Things. *IEEE Sensors Journal* 17, 20 (2017), 6816–6824.

[14] Oliver Hahm, Emmanuel Baccelli, Hauke Petersen, and Nicolas Tsiftes. 2016. Operating Systems for Low-End Devices in the Internet of Things: A Survey. *IEEE Internet of Things Journal* 3, 5 (2016), 720–734.

[15] Xabier Iturbe, Khaled Benkrid, Chuan Hong, Ali Ebrahim, Raul Torrego, and Tughrul Arslan. 2015. Microkernel Architecture and Hardware Abstraction Layer of a Reliable Reconfigurable Real-Time Operating System (R3TOS). *ACM Trans. Reconfigurable Technol. Syst.* 8, 1 (2015).

[16] Anders Blaabjerg Lange, Karsten Holm Andersen, Ulrik Pagh Schultz, and Anders Stengaard Sørensen. 2012. HartOS - a Hardware Implemented RTOS for Hard Real-time Applications. *IFAC Proceedings Volumes* 45, 7 (2012), 207–213. 11th IFAC,IEEE International Conference on Programmable Devices and Embedded Systems.

[17] James Martin. [n.d.]. *RISC-V, Rocket, and RoCC.* https://inst.eecs.berkeley.edu/~cs250/sp17/disc/lab2-disc.pdf

[18] Naotaka Maruyama, Takuya Ishikawa, Shinya Honda, Hiroaki Takada, and Katsunobu Suzuki. 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)*, 9–16.

[19] Daniel Oliveira, Miguel Costa, Sandro Pinto, and Tiago Gomes. 2020. The Future of Low-End Motes in the Internet of Things: A Prospective Paper. *Electronics* 9, 1 (2020).

[20] C. Perera, C. H. Liu, S. Jayawardena, and M. Chen. 2014. A Survey on Internet of Things From Industrial Market Perspective. *IEEE Access* 2 (2014), 1660–1679.

[21] SiFive 2016. *SiFive Freedom E300 Platform.* SiFive.

[22] Miguel Silva, David Cerdeira, Sandro Pinto, and Tiago Gomes. 2019. Operating Systems for Internet of Things Low-End Devices: Analysis and Benchmarking. *IEEE Internet of Things Journal* 6, 6 (2019), 10375–10383.

[23] Harald Sundmaeker, Patrick Guillemin, Peter Friess, and Sylvie Woelfflé. 2010. Vision and Challenges for Realizing the Internet of Things. *Cluster of European Research Projects on the Internet of Things, EU Commision* (04 2010).

[24] Bruno Sá, José Martins, and Sandro Pinto. 2021. A First Look at RISC-V Virtualization from an Embedded Systems Perspective. arXiv:2103.14951 [cs.AR]

[25] María Valdés, J.J. Rodriguez-Andina, and Milos Manic. 2017. The Internet of Things: The Role of Reconfigurable Platforms. *IEEE Industrial Electronics Magazine* 11 (09 2017), 6–19.

[26] Andrew Shell Waterman. 2016. *Design of the RISC-V instruction set architecture.* Ph.D. Dissertation. UC Berkeley.