HcM-FreeRTOS: Hardware-centric FreeRTOS for ARM Multicore

E. Qaralleh*, D. Lima**, T. Gomes**, A. Tavares**, S. Pinto**
*Princess Sumaya University for Technology
**Centro Algoritmi - University of Minho
qaralleh@psut.edu.jo
{diogo.lima, mr.gomes, atavares, sandro.pinto}@dei.uminho.pt

***Abstract*—Migration to multicore is inevitable. To harness the potential of this technology, embedded system designers need to have available operating systems (OSes) with built-in capabilities for multicore hardware. When designed to meet real-time requirements, multicore SMP (Symmetric Multiprocessing) OSes not only face the inherent problem of concurrent access to shared kernel resources, but still suffer from a bifid priority space, dictated by the co-existence of threads and interrupts.**

This work in progress paper presents the offloading of the FreeRTOS kernel components to a commercial-off-the-shelf (COTS) multicore hardware. The ARM Generic Interrupt Controller (GIC) is exploited to implement a multicore hardware-centric version of the FreeRTOS that not only solves the priority inversion problem, but also removes the need of internal software synchronization points. Promising preliminary results on performance and determinism are presented, and the research roadmap is discussed.

***Index Terms*—Unified Priority Space, RTOS, Multicore, Real-time Systems, FreeRTOS, GIC, Cortex-A9 MPCore, ARM.**

I. Introduction

Multicore technology has proven to be the only viable solution to achieve high performance without compromising power consumption [1], and its use on desktops and server environments is now ubiquitous. Over the last few years, the use of multicore processors in the embedded systems field has been growing rapidly [2], driven by the lack of performance in single-core processors to meet the demands of the current software-rich generation of embedded devices. However, in order to harness the potential of this technology, embedded system designers need to deploy applications under embedded operating systems (OSes) with built-in support for available multicore hardware [3].

OSes, in general, suffer from a bifid priority space. Threads, managed by software (kernel scheduler), need to coexist with interrupt service routines (ISRs), managed by hardware (interrupt controller). This division into thread priorities and interrupt priorities, with the latter having a higher privilege of execution, is critical on embedded real-time systems, originating a well-identified problem known as rate-monotonic priority inversion [4]. Kleyman and Eykholt have proposed the first solution [5] many years ago, and since then many other approaches have been proposed to tackle this problem [6], [7], [8], [9].

Multicore OSes, for instance, still face another problem. When designed for symmetric multiprocessing, they are conceived to satisfy two principles: (i) minimize the memory footprint; and (ii) have a full and homogeneous utilization of the processor resources. However, since the kernel data structures are present in shared memory, synchronization mechanisms for concurrent access need to be introduced internally [10]. These internal software synchronization points constitute a considerable source of indeterminism, becoming the main reason why embedded real-time operating systems (RTOSes) are delaying their transition to multicore. Only recently Müller et al. [11] addressed both aforementioned problems, extending the philosophy of the SLOTH [7] concept to the multicore domain, implemented over the AUTOSAR OS and targeting the Infineon AURIX platform.

This work in progress paper presents the implementation of a multicore hardware-centric version of the FreeRTOS, by offloading critical run-time kernel services to commercial-off-the-shelf (COTS) hardware. By exploiting the GIC of the ARM Cortex-A9 MPCore to migrate FreeRTOS system services to hardware, not only the priority inversion problem is solved, but also the need for internal software synchronization points is removed. The thread-related application programming interface (API) was kept syntactically intact to avoid the porting effort for legacy applications. Preliminary results have shown significant improvements in overall system performance and determinism.

II. Design of HcM-FreeRTOS

The rationale behind HcM-FreeRTOS is representing tasks and ISRs by abstract interrupt sources, configuring its priority and target core (Fig. 1). The system consists of (i) task activation, (ii) task dispatching and (iii) task suspending. Synchronous task activation relies on triggering the associated interrupt source via software, by writing on its respective interrupt controller register. Task dispatching, in turn, is based on saving the context (not implicitly saved by hardware) of the current executing task, followed by a branch to the new highest priority ready-to-run task. Finally, task suspending consists in forcing a task to yield its execution flow, allowing other lower priority tasks/ISRs to run. Since it is not intrinsically supported by hardware, it requires a more complex IRQ handler, which will save the context of the currently executing task in a

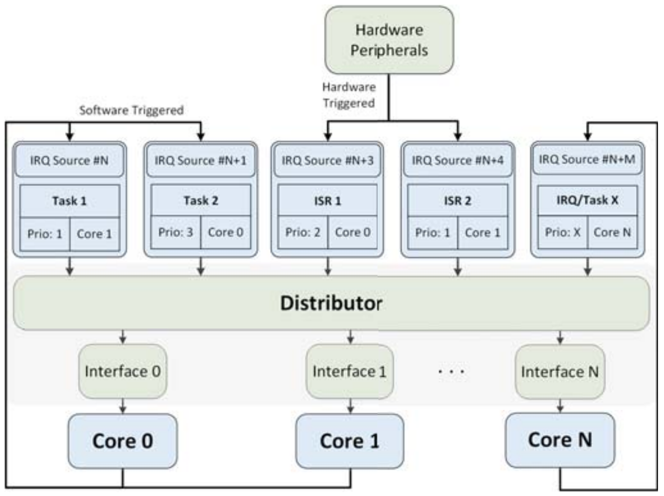

Fig. 1: Design of the hardware-centric multicore system, using interrupt handlers for the implementation of threads

dedicated stack, disable the interrupt source, and dispatch (resuming/restoring) the next ready-to-run task/ISR.

Targeting multicore platforms, HcM-FreeRTOS needs to support task activation, dispatching and suspending across multiple cores. To accomplish that, the aforementioned building blocks should be extended with a cross-core communication mechanism, typically available in the form of interprocessor interrupts.

A. Hardware Requirements

The implementation of this hardware-centric approach is only feasible if the underlying multicore hardware platform fulfils certain requirements: (i) the hardware interrupt controller must be programmable and provide several different configurable interrupt priority levels; (ii) the interrupt subsystem shall support manual triggering of interrupts as well as by software, enabling threads to be synchronously activated; (iii) the sum of the number of interrupt sources and priorities per core should be enough to cover all the threads and interrupt handlers desired for the system; (iv) a special instruction or mechanism should exist to send interrupts to remote cores.

III. Implementation of HcM-FreeRTOS

This section presents an overview of the ARM interrupt controller subsystem - GIC -, as well as a description about the HcM-FreeRTOS implementation.

A. ARM Generic Interrupt Controller

One of the COTS interrupt subsystems that fulfils the aforementioned hardware requirements is the ARM GIC, integrated in all multicore ARM Cortex-A System-on-Chips. It is partitioned into two logical blocks: the distributor and the CPU interface. The former determines the highest priority interrupt for each core and dispatches them to each CPU interface, while the latter is responsible for handling the arbitration of incoming interrupt requests locally. The GIC provides up to 1023 interrupt sources, classified in three different categories: (i) SGIs (Software Generated Interrupts) (0-15) - special interrupts generated by software for interprocessor interrupts, banked for all cores; (ii) PPIs (Private Peripheral Interrupts) (16-31) - peripheral interrupts specific to a single processor, banked for all cores; (iii) SPIs (Shared Peripheral Interrupts) (32-1023) - general interrupts shared among all cores.

B. Threads as Interrupts as Threads

As mentioned in Section II the main idea behind HcM-FreeRTOS is designing software tasks as hardware interrupts. However, the main drawback of having tasks run as pure interrupts is the run-to-completion nature of the hardware interrupt handlers. To overcome this limitation and extend interrupts to behave also as threads, a suspending feature was implemented by modelling tasks as consisting of three segments: prologue, body and epilogue [8]:

- *Prologue*: The prologue is executed wherever a high priority task is scheduled by the interrupt controller. It extends the standard behaviour of the hardware interrupt controller (i.e., it saves automatically some registers of the CPU context) by saving the remaining context of the current task, and restoring the context of the new task.
- *Body*: The body implements the task behaviour and corresponds to the application written to run in the native version of the FreeRTOS.
- *Epilogue*: The epilogue is executed wherever a task is suspended or finished. If the task was suspended, it saves the tasks context and then restores the state of the new dispatched task, otherwise the task was finished and it only restores the context of the new task.

The remaining of this section describes how the FreeRTOS task-specific system calls were re-factored. Since the ARM instruction set provides dedicated instructions that allow read and write memory atomically, no additional software synchronization points were necessary to include in order to deal with concurrent access to the GIC distributor registers.

1) Scheduler Start: Starting the scheduler is fairly straightforward, consisting in enabling the GIC distributor and each CPU interface through the `GICD_CTRL` and `GICC_CTRL` registers, respectively.

2) Task Creation/Activation: Whenever a task is created, the existing TCB structure is initialized (allocating the task stack), and the associated interrupt is configured. Thereby, the GIC distributor requires specifying the priority level (`GICD_IPRIORITYRx`), setting the target CPU (`GICD_ITARGETSRx`) - following a round robin schema -, linking the interrupt handler to the task-specific code, enabling (`GICD_ISENABLERx`) and setting the interrupt as pending (`GICD_ISPENDRx`). After, locally to each CPU interface, if the created task has higher priority than the currently executing task, the prologue is executed and the created task dispatched, hence no SGI (i.e., cross-core interaction) is needed.

3) Task Deletion: Whenever a task is deleted the interrupt source linked to that task is disabled (`GICD_ICENABLERx`), it is signalled as waiting deletion (to free the memory during the idle periods), and if the task is currently executing in the local core then the epilogue is performed, otherwise an SGI is sent (i.e., cross-core interaction) to signal the remote core to delete the current task.

4) Task Suspend: Whenever a task is suspended the pending flag of the interrupt source linked to that task is disabled (`GICD_ICPENDRx`), and if the task is currently executing in the local core the epilogue is performed, otherwise an SGI is sent (i.e., cross-core interaction) to signal the remote core to suspend the current task.

5) Task Resume: Whenever a task is resumed its state is changed to ready and the interrupt is set as pending (`GICD_ISPENDRx`). After, locally to each CPU interface, if the resumed task has higher priority than the currently executing task, the prologue is executed and the resumed task dispatched.

IV. Preliminary Results

The implemented solution was tested on the Fast Models emulator, using a model of the Versatile Express (VE) board with a dual- and quad-core ARM Cortex-A9. We compared the native single-core version of the FreeRTOS (ver. 7.0.2) against our redesigned hardware-centric multicore version (HcM-FreeRTOS). For our implementation we experimented and gathered results also for cross-core interaction. The results were obtained using the Performance Monitoring Unit (PMU) component, and the software was compiled with the ARM Xilinx Toolchain.

In order to assess the performance and determinism results we performed several microbenchmarks. The selected microbenchmarks encompass the modified system calls, which include:

- `xTaskCreate`: Creates a task and dispatches it if its priority is higher than the currently executing task;
- `vTaskDelete`: Deletes a task and dispatches another if the deleted task is currently executing;
- `vTaskSuspend`: Suspends a task and dispatches another if the suspended task is currently executing;
- `vTaskResume`: Resumes a task and dispatches it if its priority is higher than the currently executing task;
- `vTaskSetPriority`: Changes the priority of a task and dispatches it if the modified priority is higher than the priority of the currently executing task;

For each microbenchmark we performed several experiments with different system configurations, changing parameters such as: (i) the number of tasks (from 1 to 32); (ii) the priority of tasks (from 1 to 255); (iii) the number of tasks with the same priority (from 1 to 3); (iv) the priority gap between tasks (from 32 to 253), and (v) the priority of the dispatched or not dispatched task. The presented results report the mean value ($\overline{x}$) and the standard deviation (s) of a set of experiments.

We start by performing the behaviour evaluation, by extending the aforementioned test scenarios with distinct priority levels of hardware interrupts. It is naturally perceptible that we solved the rate monotonic priority inversion problem by design, and we effectively corroborated our predictions by observing an unified execution flow, with tasks and ISRs coexisting correctly. During all experiments no ISR with semantically low priority has interrupted a task with higher priority, fact that was not observed in the native version of FreeRTOS.

API	***Dispatch***		**FreeRTOS**		**HcM**		
	IC	***CC***	$\overline{x}$	***s***	$\overline{x}$	***s***	***ov.(%)***
xTaskCreate	w	-	1089	2	1042	0	***-4.3***
	w/o	-	968	0	945	0	***-2.4***
	-	w	-	-	1042	0	-
	-	w/o	-	-	945	0	-
vTaskDelete	w	-	2955	2891	306	0	***-89.6***
	w/o	-	187	17	168	0	***-10.2***
	-	w	-	-	571	0	-
	-	w/o	-	-	168	0	-
vTaskSuspend	w	-	2941	2891	283	0	***-90.4***
	w/o	-	158	2	81	0	***-48.6***
	-	w	-	-	446	0	-
	-	w/o	-	-	81	0	-
vTaskResume	w	-	301	2	152	0	***-49.6***
	w/o	-	207	0	51	0	***-75.4***
	-	w	-	-	152	0	-
	-	w/o	-	-	51	0	-
vTaskPrioritySet	w	-	2976	2884	170	0	***-94.3***
	w/o	-	278	64	74	0	***-73.3***
	-	w	-	-	170	0	-
	-	w/o	-	-	74	0	-

TABLE I: Performance and Determinism Evaluation Results

Table I, in turn, presents the achieved results for the overhead evaluation. It is clear that our implementation overcomes the native version of FreeRTOS in both metrics: performance and determinism. Relatively to the former, the execution time was reduced between 2.4% (`xTaskCreate` without dispatch) and 94.3% (`vTaskPrioritySet` with dispatch). The speedup achieved in the `xTaskCreate` API is considerably smaller than the other APIs, because we implemented the suspend feature and so, it still requires stack allocation for each task. Regarding determinism, our evaluation methodology outlined important sources of indeterminism in FreeRTOS, stemming from its searching algorithm - that determines the next running task - in the context switch operation. Since our approach is based on hardware and relies on the GIC to provide the highest priority ready-to-run task, the context switch, in particular, and the APIs, in general, are naturally deterministic. This is why our approach achieved a null standard deviation in all experiments.

The results still corroborated the viability of implementing multicore RTOS without the need of software synchronization mechanisms for the concurrent access to shared kernel resources. We only needed to guarantee the coherency during GIC distributor registers accesses, and we did that with specific and dedicated hardware instructions. Determinism was not compromised and the only extra overhead with the advent of multicore migration came from cross-core interaction, due to the need to trigger interprocessor interrrupts (SGIs).

V. Research Roadmap

Work in the near future will proceed through the migration of the remaining kernel services to hardware. At this stage, only a subset of the task management API is exploiting the hardware interrupt controller to implement the scheduling decisions. However, the idea is not only offload to hardware all the scheduling services, but also implement other kernel services. For example, synchronization mechanisms such as mutual exclusion (mutex), can easily be implemented exploiting the GIC. For local resources (resources shared between the same processor) the Priority Ceiling Protocol based on the temporary raise of tasks' priority will be implemented [8]. For global resources (resources shared between different processors), we will investigate the applicability of the Multiprocessor Priority Ceiling Protocol [12].

After migrating all kernel services, research will focus on an in-depth and real-world system evaluation. First, we will focus on performance and indeterminism. All the kernel services will be evaluated, not only by performing microbenchmarks, but also running concrete benchmark suites - Thread Metrics and MiBench Suites are good candidates. More sources of indeterminism will be also investigated, and characterized. Furthermore, experiments will be carried out in a physical multicore development platform (e.g., Xilinx Zynq ZC702) to assess real-world results, because Fast Models is an excellent tool for proof-of-concept but does not model accurate cycle counts - all instructions take the same amount of time. Memory footprint and code management (maintainability) are other metrics that will be also evaluated.

From a different perspective, research will continue towards the development of an efficient hardware-based dual-OS architecture. With the emergent complexity of modern embedded devices, which increasingly demand for general purpose computing characteristics but still need to guarantee the real-time requirements, it is necessary to develop efficient solutions that allow the coexistence of General Purpose Operating Systems (GPOSes) with RTOSes. Thereby, the solution of our previous work with ARM TrustZone technology [13] will be followed to provide the spatial and temporal isolation between the OSes. Moreover, we will go one step further applying concepts of this work to the RTOS running on the secure side. By exploiting only ARM COTS hardware, we will provide efficiency and optimization at two different levels of the system stack: not only on the virtualization layer but also in the OS layer.

VI. Conclusion

Over the last few years, the interest in embedded multicore systems has increased significantly due to the simultaneous advantages on power and performance. However, embedded RTOSes with built-in multicore support face two well identified problems: the need of internal software synchronization points, and the lack of an unified priority space. This paper presented a work in progress towards the implementation of a multicore hardware-centric version of the FreeRTOS, by offloading some kernel components to COTS hardware. Migrating the scheduling decisions to the interrupt controller we showed that it is possible to overcome the problems of multicore RTOSes and simultaneously improve performance and specially determinism.

The research roadmap section described that research in the near future will focus on the migration of the remaining kernel services to hardware, and on an extensive system evaluation on a real multicore platform. Research will then proceed towards the development of an efficient hardware-centric dual-OS architecture, by exploiting only COTS ARM SoC capabilities.

VII. Acknowledgement

Sandro Pinto is supported by FCT - Fundação para a Ciência e Tecnologia (grant SFRH/BD/91530/2012). This work has been supported by FCT - Fundação para a Ciência e Tecnologia within the Project Scope: PEst-UID/CEC/00319/2013.

References

[1] M. Karlsson, "Enea Hypervisor: Facilitating Multicore Migration with the Enea Hypervisor," *ENEA White Paper*, pp. 1–11, 2012.

[2] F. Reichenbach and A. Wold, "Multi-core technology – next evolution step in safety critical systems for industrial applications?" in *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference on*, Sept 2010, pp. 339–346.

[3] D. Andrews, I. Bate, T. Nolte, C. Otero-Perez, and S. M. Petters, "Impact of embedded systems evolution on rtos use and design," in *1st International Workshop Operating System Platforms for Embedded Real-Time Applications (OSPERT'05)*, 2005.

[4] L. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz, "Predictable interrupt management for real time kernels over conventional pc hardware," in *Real-Time and Embedded Technology and Applications Symposium, 2006. Proceedings of the 12th IEEE*, April 2006, pp. 14–23.

[5] S. Kleiman and J. Eykholt, "Interrupts as threads," *SIGOPS Oper. Syst. Rev.*, vol. 29, no. 2, pp. 21–26, Apr. 1995.

[6] L. Leyva-del Foyo, P. Mejia-Alvarez, and D. de Niz, "Predictable interrupt scheduling with low overhead for real-time kernels," in *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, 2006, pp. 385–394.

[7] W. Hofer, D. Lohmann, and W. Schroder-Preikschat, "Sleepy sloth: Threads as interrupts as threads," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, Nov 2011, pp. 67–77.

[8] S. Pinto, J. Pereira, D. Oliveira, F. Alves, E. Qaralleh, M. Ekpanyapong, J. Cabral, and A. Tavares, "Porting sloth system to FreeRTOS running on ARM cortex-m3," in *Industrial Electronics (ISIE), 2014 IEEE 23rd International Symposium on*, June 2014, pp. 1888–1893.

[9] T. Gomes, P. Garcia, F. Salgado, J. Monteiro, M. Ekpanyapong, and A. Tavares, "Task-aware interrupt controller: Priority space unification in real-time systems," *Embedded Systems Letters, IEEE*, vol. 7, no. 1, pp. 27–30, March 2015.

[10] J. Mistry, M. Naylor, and J. Woodcock, "Adapting freertos for multicores: an experience report," *Software: Practice and Experience*, vol. 44, no. 9, pp. 1129–1154, 2014.

[11] R. Müller, D. Danner, W. Preikschat, and D. Lohmann, "Multi sloth: An efficient multi-core rtos using hardware-based scheduling," in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, July 2014, pp. 189–198.

[12] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, May 1990, pp. 116–123.

[13] S. Pinto, D. Oliveira, J. Pereira, N. Cardoso, M. Ekpanyapong, J. Cabral, and A. Tavares, "Towards a lightweight embedded virtualization architecture exploiting arm trustzone," in *Emerging Technology and Factory Automation (ETFA), 2014 IEEE*, Sept 2014, pp. 1–4.