Co-Designed FreeRTOS Deployed on FPGA

J. Pereira, D. Oliveira, S. Pinto, N. Cardoso, V. Silva, T. Gomes, J. Mendes and P. Cardoso
Centro Algoritmi - University of Minho
{jorge.m.pereira, daniel.oliveira, sandro.pinto, vitor.alberto, tiago.a.gomes}@algoritmi.uminho.pt
{ncardoso, jose.mendes, paulo.cardoso}@dei.uminho.pt

***Abstract*—Most embedded systems are bound to real-time constraints. Two of the critical metrics presented in these systems are determinism and latency. Due to growing in complexity of embedded applications, real time operating systems (RTOS) are needed, not only to hide the increasingly complex hardware, but also to provide services to the system's running tasks. Unfortunately, this new layer on an embedded system puts more pressure on the aforementioned metrics. One of the ways to cope with this problem is to offload RTOS run-time services to the hardware layer.**

This paper presents a hybrid hardware/software implementation of this technique upon the well known FreeRTOS, improving system's latency and predictability, by migrating critical run-time services to hardware. The developed hardware accelerators were synthesized on a field-programmable gate array (FPGA), exploiting the point-to-point bus Fast Simplex Link (FSL) to interconnect to the Xilinx's Microbaze soft-core processor.

***Index Terms*—Real-time Systems, Determinism, Latency, FreeRTOS, Hardware Accelerators.**

I. Introduction

Real-time systems are typically present in embedded devices being designed to support time constrained tasks in meeting their deadlines. Depending on the need to meet strict deadlines, these systems can be classified in hard or soft real time [1]. In hard real-time applications, determinism and latency are a critical metrics, since missing deadlines may result catastrophic for the purposes the system serves. In contrast, for soft real-time systems, missing a deadline is normally not as critical as in the hard real-time systems.

The presence of a real-time operating system (RTOS) introduces new sources of latency and lack of determinism. Latency is "wasted" time and its minimization means better responsivity of the RTOS. This paper addresses a specific source of software latency, the RTOS lists handling. Lack of determinism is caused by response time variation (jitter), another of the least desired characteristics of a RTOS. Most of jitter sources comes from RTOS' dynamic data structures and their management and traversal. For example, the time taken to select next task to run in the ready queue list of FreeRTOS priority-based scheduler, depends on its position in the list. The same reasoning can be applied to other RTOS list-like structures, namely the timer or mutex wait lists.

In this paper we present a hybrid hardware/software RTOS that takes advantage of hardware accelerators to improve these critical metrics and, consequently, improve overall system performance. Therefore, our approach demonstrates that co-designed RTOS has a better performance and most importantly maintains consistent response time and more predictability compared to its purely software version.

The organization of the paper is as follows: Section II presents related work. Section III explains the system architecture, detailing the operating system, the soft-core processor, the communication mechanism and are also outlined the software parts that were migrate to hardware and the reasoning behind that migration. Section IV describes the hardware accelerators, how they work and implementation details. Section V discusses the experimental results. Lastly, conclusions and future work are presented in section VI.

II. Related Work

RTOSs nowadays face requirements such as predictability and low latency, not achieved with more computational power. Migration to hardware of software tasks and services, addresses these issues leading to solutions able to cope with these increasingly strict requirements. Migration of RTOS services, such as scheduling, time management and task management, to dedicated hardware modules, provides increased system performance and allows the RTOS to meet metrics requirements [2]. The well known concept of software thread, similar as the POSIX model [3], can be applied and shift the paradigm HW thread, providing a unified transparent model [3]. The coexistence of a hybrid, software and hardware model in an operating system environment, raises concerns namely regarding an unified programming model, portability, legacy software support, suitable interface and synchronization mechanisms, communication overhead and resource optimization [4], [5], [6], eventually exacerbated in resource constrained embedded contexts. As an example of an unified hardware/software thread programming model, [4], [5], show some of the benefis of this methodology. Communication overhead on a shared bus, and resource utilization are maintained low in both cases, while still offering performance improvements. [5], [7] ,[6] also argue on the importance of a hardware/software transparent model and homogenous interfaces, in order to achieve suitable HW abstraction and legacy-software reutilization.

Aiming to promote the exploitation of FPGA-based embedded systems performance, [8] presents the foundations for an HW accelerated RTOS with run-time partial reconfiguration. Regardless targeting high-end embedded systems, i.e. not resource constrained, "hthreads" Real time Kernel [9], and ReconOS [10], are examples of this approach. These systems can meet requirements otherwise very difficult to achieve on a software-only RTOS, providing llow latency, low jitter, in a

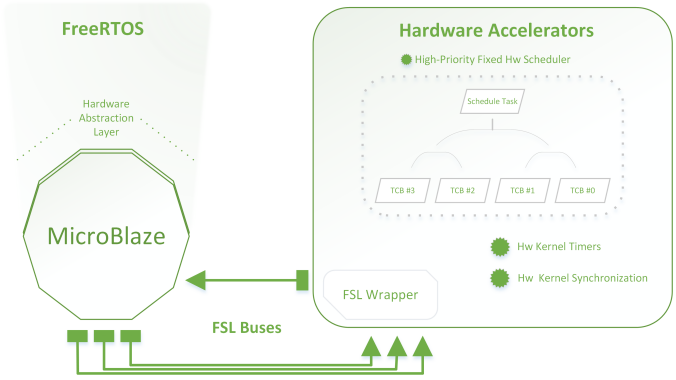

Fig. 1: System Architecture Overview

true parallell system that scales independently of the number of system tasks.

III. Architecture

A. Overview

Figure 1 depicts the architecture of the hybrid FreeRTOS. It consists in three main hardware components synthesized in the FPGA fabric: (i) the soft-core processor (MicroBlaze) where the FreeRTOS and its application runs, above the hardware abstraction layer (HAL); (ii) the developed hardware accelerators intended to improve the predictability and latency of FreeRTOS; and (iii) the FSL buses, used to interconnect the hardware accelerators to the MicroBlaze soft-core.

Moreover, there are other hardware components synthesized in the FPGA in order to support the RTOS system tick, which includes a interrupt controller and a timer, both provided by Xilinx's IP library. The connection between these two IP's and the soft-core processor is done by PLB bus, which implements a master/slave communication model.

B. Operating System and Soft-core Processor

1) FreeRTOS: FreeRTOS [11], [12] is a real-time operating system targeting low-end embedded systems with limited resources supporting thirty-four different architectures. Besides, its source code architecture is designed in order to enhance its portability, being mainly composed by two layers, an hardware independent layer and a portable layer.

The RTOS kernel can be tailored to the application being built through a configuration file called FreeRTOSConfig.h, where it is possible to adjust clock speed, heap size, mutual exclusion objects, API subsets, etc. Moreover, being a open-source RTOS with a small and straightforward kernel, makes it possible to a effortless internal redesign. These set of advantages and features justified the use of FreeRTOS as the target RTOS for this work.

2) MicroBlaze: MicroBlaze is a soft-core embedded processor designed and optimized for implementation in Xilinx FPGAs [13]. This soft-core processor is supported by the most recent Xilinx FPGAs in particular Virtex-7, Kintex-7, Artix-7 and the Xilinx Zynq-7000. MicroBlaze core is organized as a 32-bit Harvard load/store architecture with separated bus interface units for data and instruction accesses. It provides four different memory interfaces: (i) Local Memory Bus (LMB) to access on-chip local-memory; (ii) IBM Processor Local Bus (PLB) or the AMBA AXI4 (AXI4) to connect on-chip and off-chip peripherals and memory; (iii) Xilinx CacheLink (XCL) to interface with high performance specialized external memory controllers; and (iv) Fast Simplex Link (FSL) or AXI4-Stream interface to provide a fast non-arbitrated streaming communication mechanism.

Finally, the MicroBlaze is highly configurable, providing selective enabling of additional functionality, namely in terms of cache size, pipeline depth, integrated peripherals, MMU and bus-interfaces.

3) Fast Simplex Link Bus: The Fast Simplex Link (FSL) bus is a fast communication mechanism between two design elements [14] and is structured with a unidirectional point-to-point FIFO-based communication, which can be configured as master or slave. With a maximum transfer speed of 300 million words/sec, the coprocessors connected to the master ports of the FSL bus pushes data and control signals onto the FIFO. In contrast, the slave ports of the FSL bus reads and pops from the FIFO buffer. The internal clock of this buffer has the same frequency of the processor system clock.

MicroBlaze offers 16 parallel FSL channels, equally distributed between masters and slaves ports. In this paper, the FSL bus was chosen to interconnect the hardware accelerators to the soft-core, since it provides, as previously mentioned, point-to-point dedicated channels, which each accelerator can take advantage of, overcoming the traditional problem of concurrent access and avoiding latency problems, regarding other buses (e.g. PLB bus).

C. Selection of the dedicated hardware accelerators

The FreeRTOS components that have been migrated from the software layer to the hardware layer were the scheduler and the kernel software timers. These two components represent the services available on FreeRTOS that are the major sources of jitter and overhead.

The heart of an operating system is the scheduler. It selects the ready task to execute next, based on its scheduling algorithm. In a complex and heavy system, where there are dozens of tasks ready to execute, the scheduler can introduce undesired response jitter and a lot of overhead. Taking this into account, the FreeRTOS scheduler was off-loaded into the FPGA fabric.

Delaying tasks is also typically needed in an embedded system. FreeRTOS achieves this by providing for each task a dedicated software-based counter, that is decremented at each system tick. In a complex system, where it is likely to exist several delayed tasks, this procedure could represent a huge source of jitter. Therefore, it is advantageous to free the kernel of these software-based counters by migrating them to the hardware layer.

The FreeRTOS API was maintained intact eliminating the porting effort for legacy applications. Therefore, migration of FreeRTOS run-time services to the hardware involved changes

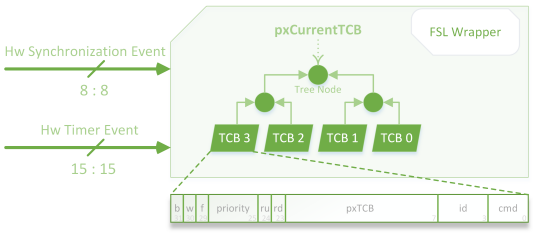

Fig. 2: Hardware scheduler overview

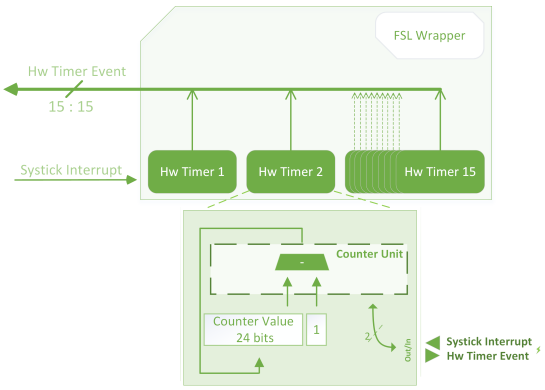

Fig. 3: Hardware Kernel Timers overview

to the body of several functions in hardware abstraction layer (HAL) by calling the respective hardware modules.

Migration of the aforementioned FreeRTOS services into the hardware layer are responsible for the reduction of jitter and system overhead, increasing the RTOS determinism and predictability, which will be presented later in this paper.

IV. Implementation

A. Hardware Scheduler

As previously mentioned, the RTOS scheduler is a large time consuming mechanism and one of the sources of jitter. Hence, the overhead introduced, can lead to the miss of deadlines, which in the case of hard real-time systems could have catastrophic consequences. One of the solutions to reduce such overhead is accomplished by migrating it to hardware.

The implemented hardware scheduler is depicted in Figure 2 and has the following relevant features: (i) the selection algorithm is based on a fixed-priority pre-emptive scheduling; (ii) the data structure architecture is based on a binary tree structure; (iii) the number of supported tasks is parametrized in a factor of 2^n; (iv) the information about the tasks in kept on an on-chip register bank; (v) the implementation is based on combinational logic, which enables the scheduler to provide the highest priority task in a single clock cycle; (vi) and finally, the communication between this hardware accelerator and the CPU is performed by the FSL bus, through a data packet of 32-bits width.

As said, the core data structure is a binary tree. The base HDL module (`TreeNode2`) of the binary tree (for N=1) is composed by a node with the combinational logic expression to find the most priority task between two tasks. The other modules will be composed always by a `TreeNode2` and another two modules equal to the tree node for the factor N-1. This means, for example, that the module for N=2 will be composed by three base modules (two N-1 `TreeNode2`'s plus another `TreeNode2`). Since all the modules are built using combinational logic, the time needed to find the most priority task is equal to the propagation time of the signals, which is a huge improvement in contrast with the software counterpart.

In Figure 2 is presented the hardware task tree and the data contained in each register representing a task in hardware. The three most significant bits represent the current state of the task ("b": blocked, "w": suspended or "f": finished respectively); "priority" corresponds, as the name implies, to the priority level of the task; "run" and "ready" are also used to define the tasks state; the "pxTCB" field contains the pointer to the address of the software TCB; "ID" identifies the task in this register bank; and finally, "command" represent the operation to be performed (e.g. read the most priority task or write the task into the register bank).

The FSL wrapper implements a simple protocol, containing two messages, "read" and "write". When a read message is received in the module, this acknowledges sending the most priority task. When a write message is received, it is followed by the data to be inserted in the respective register.

B. Hardware Kernel Timers

FreeRTOS provides services to block a task for a given number of ticks. If one of these services is called, the RTOS has a dedicated software timer, which is related to each delayed task. At each system tick, the software timer is decremented and verified if it has expired, and if this is the case, the task is unblocked. This type of operation injects jitter in the system and consequently non-determinism, because the time to update each counter depends on the number of the delayed tasks present in the system. In a hardware approach, and due to the parallel nature of the hardware, it is possible to reduce and even eliminate this source of non-determinism present in the operating system.

Figure 3 depicts the internal block diagram of the hardware kernel timers' IP. It has a bank of n timers, where the n is the number of tasks accepted by the scheduler. Each timer, as in the software-based version, is related to a task.

This component is connected to the MicroBlaze and to the scheduler IP. The connection between the processor and this peripheral is done only by the FSL master bus, because it is a write-only peripheral. The output of this device is hard-wired directly to the scheduler. Whenever a timer reaches zero, the associated task must change from suspended state to ready state. This is done, by sending a signal from the hardware kernel timer to the scheduler.

In addition to the timer bank, this IP has also a module in charge of decoding the data packet from the processor, delivered through the FSL bus. This data packet is composed by three fields. The lower twenty four bits, are the value of the desired suspension time for the task identified in the next five

bits. The remaining bits out of the 32-bits data packet are not defined, and could be used in further versions of this project.

Each timer is independent from the others, so the operations executed on one of them does not interfere with the others. Besides that, each is hard-wired with a specific task on the scheduler, so even if all the timers expire at the same time, the time needed to process the changes at the scheduler side is always the same. Thereby, the jitter and non-determinism caused by the process of transition between states (from suspended to ready), introduced by software counterparts, are also eliminated.

As in the hardware scheduler, an FSL wrapper was required to establish the communication between this IP and the MicroBlaze. In this case is only required to give support to the write operation. When a write message is received with the intended configuration, the counter is programmed and it will start to decrement at each system tick.

C. Hardware Abstraction Layer

Leveraging accelerators offloading, requires re-factoring of FreeRTOS hardware abstraction layer. The main changes will target FreeRTOS functions related to task control and management (i.e. `vTaskCreate()`, `vTaskSuspend()`, `vTaskResume()`, `vTaskDelay()` and `vTaskDelete()`).

Task creation (`vTaskCreate()`) needs to generate an ID, which will identify the task in the hardware scheduler, and send an FSL data packet to the hardware scheduler with all needed information (ID, the pointer to the TCB and the task priority). It should be noted that the aforementioned ID is a new field added to the TCBs structure. When a task is deleted (`vTaskDelete()`) a data packet is sent to the hardware scheduler, changing only the finish bit in the corresponding task TCB entry.

FreeRTOS allows the suspension of a task (`vTaskSuspend()`), for an indefinite amount of time, and after its resumption (`TaskResume()`). Besides that, it is possible to suspend a task for a fixed period (`vTaskDelay()`). Suspending a task in the context of the hybrid FreeRTOS, is nothing more than sending a data packet through the FSL bus to the hardware scheduler, where the blocked bit is set and the others cleared. On the other hand, resuming is also very straightforward, since it reverts the previous settings. At last, to delay a task, instead of communicating with the scheduler, a data packet is sent to the hardware timers block, which is then responsible to signal the changes to the hardware scheduler.

Finally, is also relevant to say that a complete redesign was made to the FreeRTOS function responsible for context switching (`vTaskSwitchContext()`). This procedure consists in reading the hardware scheduler through its FSL-based bus to get the next ready task. The received data packet is decoded, by performing a mask, to retrieve the pointer to the TCB of the new task to execute. At last, the `pxCurrentTCB` is updated to point to the new task's TCB.

V. Experimental Results

The presented system architecture was implemented on a XUP Virtex-II Pro development platform under MicroBlaze 7.10d and FreeRTOS 7.4.0 versions, and the project was developed and synthesized using the EDK 10.1 tools ecosystem, provided by Xilinx. The implementation of the dedicated hardware modules was done in Verilog.

In the following section are presented the results obtained from measurements made on the hybrid RTOS, compared to the software-based RTOS.

A. Evaluation

The hardware accelerators implemented to support FreeRTOS are directly related to the task switching mechanism. This way, to infer the benefits obtained from the hardware approach, the evaluation and validation was realized by measuring the latency and jitter in the manipulation of the various kernel data structures.

In order to access the improvements introduced by our approach, both the hybrid and the software-based versions were instrumented through a specific hardware counter module, built for this purpose. This hardware module does not interfere with the system and takes advantage of an FSL interface.

```
void vTickISR( void *pvBaseAddress ){

            CONFIG_COUNT();
            START_COUNT();

#ifdef (Hybrid_FreeRTOS == 0)
            vTaskIncrementTick();
#endif

            STOP_COUNT();
            READ_COUNT();
            CLEAR_COUNT();

            (...)

            START_COUNT();

            vTaskSwitchContext();

            STOP_COUNT();
            READ_COUNT();
            CLEAR_COUNT();

}
```

Listing 1: Evaluation measure points

In the Listing 1, shows that the software-based version measurements were taken in two points: (i) during a scheduling point, specifically the switching between tasks (`vTaskSwitchContext()`) and (ii) the increment of the software timers (`vTaskIncrementTick()`) executed at each system tick (`vTickISR()`). On the other hand, in the case of the hybrid version, as the software timers were migrated to hardware, so there is no software, measurements were taken only during the scheduling point and the corresponding new scheduled task (`vTaskSwitchContext()`). Table I show the results at `vTaskSwitchContext()`, and Table II shows the results at `vTaskIncrementTick()`.

TABLE I: Measured latency during the execution of `vTaskContextSwitch()`.

	5 Tasks		10 Tasks		15 Tasks	
	FreeRTOS (Sw)	FreeRTOS (Hw)	FreeRTOS (Sw)	FreeRTOS (Hw)	FreeRTOS (Sw)	FreeRTOS (Hw)
μ	86.51	46.00	89.31	46.00	89.90	46.00
σ	13.46	0.00	14.64	0.00	14.78	0.00

TABLE II: Measured latency during the execution of `vTaskIncrementTick()`.

	5 Tasks	10 Tasks	15 Tasks
	FreeRTOS (Sw)	FreeRTOS (Sw)	FreeRTOS (Sw)
μ	305.97	359.87	375.90
σ	212.67	259.72	259.75

In Table I are presented the results taken from three different application scenarios. All of them are based in the same control flow: a number of tasks (five, ten and fifteen) are created before the start of the scheduler (`vTaskStartScheduler`) and they have atomic priority levels; when the scheduler starts, Task1 will be automatically dispatched and after this task will suspend itself, leading to the dispatch of Task2, and so on.

As can be seen from the collected data, the mean latency in the hybrid version of FreeRTOS is significant lower than the value from the software-based version. More than that, regardless the number of tasks within the system, this operation takes 46 system ticks. As a result, the standard deviation, which by itself represents the presence of jitter, is reduced to zero in the hybrid approach.

During the measurements, was possible to verify that the increment of the counters associated with each task is a costly and time consuming procedure, as well as one of the biggest sources of jitter. Since, in our approach this procedure was completely moved to hardware, the RTOS will be entirely free of this jitter and latency. Table II depicts the results obtained just from the software-based RTOS for `vTaskIncrementTick`, due to previously mentioned fact that for the hybrid RTOS these results will be zero.

VI. Conclusion and Future Work

In this paper, we presented a hybrid real-time operating system based on FreeRTOS to take advantage of the developed hardware accelerators to improve system latency and determinism. This approach led to a internally redesign of FreeRTOS software, more specifically inside the hardware abstraction layer, in terms of the functions responsible for the management and synchronization of tasks.

The evaluation results showed convincing improvements on the aimed levels: system latency and determinism. The execution time to schedule a new task was reduced on average at least by 53.17%, 51.50% and 51.16% respectively for each test-case, when compared with the software-based version. More important, the execution time for managing tasks is fixed, regardless the number of tasks, which proves the improvement in system predictability. Also, the jitter and latency present in FreeRTOS due to the software timers was also removed with the introduction of the hardware kernel timers in the system.

In short, in the era that embedded systems are rather complex and continuously evolving, real-time operating systems, such FreeRTOS, require a more deterministic and predictable execution. Thereby, implementing software components in hardware accelerators can improve system performance and reduce response jitter, as ours experimental results proved.

Proposed as future work is the redesign of a full version of FreeRTOS following our approach, aiming the complete support and refactoring of all the remaining APIs. Moreover, the hardware scheduler should be redesigned in order to prevent the priority inversion scenarios that occurs when tasks and interrupts coexist in the same system. Additionally, considering that multiprocessing is becoming an emergent trend on todays embedded market, future research will be also focused on ways and possibilities to migrate our approach to multi-core architectures.

Acknowledgment

This work has been supported by FCT - Foundation for Science and Technology within the Project Scope: PEst-OE/EEI/UI0319/2014.

References

[1] R. Oshana and M. Kraeling, *Software Engineering for Embedded Systems: Methods, Practical Techniques, and Applications (Expert Guide)*, Newnes, Ed., May 2013.

[2] J. Furunäs, "Benchmarking of a real-time system that utilises a booster," *International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 2015–2022, June 2000.

[3] D. Thomas, J. Adams, and H. Schmit, "A model and methodology for hardware-software codesign," *Design & Test of Computers, IEEE*, vol. 10, no. 3, pp. 6–15, 1993.

[4] B. Zhou, W. Qiu, Y. Chen, and C. Peng, "SHUM-uCOS: A RTOS using multitask model to reduce migration cost between SW/HW tasks," *Proceedings of the 9th International Conference, Computer Supported Cooperative Work in Design*, vol. 2, pp. 984–989, May 2005.

[5] Y. Wang, W. Chen, X. Wang, H. You, and C. Peng, "The hardware thread interface design and adaptation on dynamically reconfigurable soc," *International Conference on Embedded Software and Systems.*, pp. 173–178, May 2009.

[6] S. Nordstrom and L. Asplund, "Configurable hardware/software support for single processor real time kernels," *International Symposium on System-on-Chip.*, pp. 1–4, November 2007.

[7] H. K. So, A. Tkachenko, and R. Brodersen, "Hardware microkernels a novel method for constructing operating systems for heterogeneous multi core platforms," *Proceedings of the 4th international conference on Hardware/software codesign and system synthesis.*, pp. 259–264, May 2006.

[8] X. Iturbe, K. Benkrid, A. T. Erdogan, T. Arslan, M. Azkarate, I. Martinez, and A. Perez, "R3tos: A reliable reconfigurable real-time operating system," *NASA/ESA Conference on Adaptive Hardware and Systems.*, pp. 15–18, June 2010.

[9] J. Agron, W. Peck, E. A. amd D. Andrews, E. Komp, R. Sass, F. Baijot, J. Stevens, and I. H. Road, "Run-time services for hybrid cpu/fpga systems on chip," *27th IEEE International on Real Time Systems Symposium.*, pp. 3–12, December 2006.

[10] E. Lübbers and M. Platzner, "ReconOS: An RTOS supporting hard- and software threads," *17th International Conference on Field Programmable Logic and Applications.*, pp. 441–446, August 2007.

[11] "Freertos homepage." [Online]. Available: http://www.freertos.org/

[12] R. Barry, *Using the FreeRTOS Real Time Kernel*, 1st ed., 2010.

[13] *LogiCORE IP MicroBlaze Micro Controller System (v1.1)*, Xilinx.

[14] *LogiCORE IP Fast Simplex Link (FSL) V20 Bus (v2.11c)*, Xilinx.