Hardware Co-Processor for the OReK Real-Time Executive

Carlos Miguel Ferreira, Arnaldo S. R. Oliveira

***Resumo* – Este artigo discute os benefícios em utilizar um co-processador, para melhorar o determinismo e desempenho de um executivo de tempo-real. O co-processador proposto foi modelado numa linguagem de descrição de *hardware* (VHDL) e implementado num circuito reconfigurável (FPGA - *Field Programmable Gate Array*). Tem a capacidade de gerir (escalonamento, preempção e execução) várias tarefas tanto periódicas como aperiódicas. A preempção de uma tarefa que executa na Unidade de Processamento Central (CPU) é feita através de uma linha de pedidos de interrupção, que liga o co-processador ao CPU. Fontes externas de interrupção são ligadas ao co-processador para permitir uma activação controlada e despacho das respectivas tarefas de serviço.**

A validação e avaliação do executivo de tempo-real com a unidade de co-processamento, mostra um aumento significativo do determinismo e desempenho do sistema, quando comparado com o mesmo sistema mas sem a ajuda da unidade de co-processamento, ou seja, executando por completo em software.

***Palavras chave* – Acelerador de hardware, sistemas de tempo-real, co-processador, FPGA, desempenho determinístico**

***Abstract* – This paper discusses the benefits of using a hardware coprocessor to improve the determinism and performance of a Real-Time Kernel. The proposed coprocessor was modeled with the VHDL hardware description language and implemented in a FPGA (Field-Programmable Gate Array). It is able to manage (schedule, preempt and dispatch) several tasks, either periodic or aperiodic. The preemption of the task running on the Central Processing Unit (CPU) is performed through an interrupt line that connects the coprocessor to the CPU. External interrupt sources are connected to the coprocessor to allow a controlled activation and dispatching of the respective service tasks.**

The validation and benchmarking of the real-time kernel with the co-processing unit, shows a significant increase on the determinism and performance of the system, when compared with the same system but without the help of the co-processing unit, i.e. running fully in software.

***Keywords* – RTOS hardware accelerator, FPGA-based coprocessor, deterministic performance**

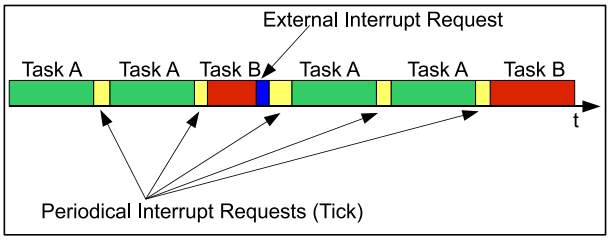
I. INTRODUCTION

Traditional Embedded systems are generally composed by a Central Processing Unit, Memory and Input/Output devices. In order to manage all the resources available, a Real-Time Kernel is normally employed. Software kernels are generally preferred, simply because they are more easy to develop, to test and upgrade every time a new version is released.

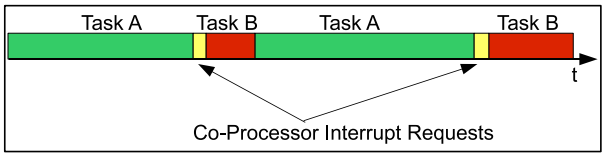
Real-Time kernels diverge from the traditional general purpose kernels. While the main objective of a general purpose kernel, is to reduce the applications tasks response time while giving to each one a fair use of the CPU, the goal of a real-time kernel is to be deterministic, i.e. to guarantee a maximum response time to external events and to maintain the tasks activation and execution, within a rigorous scheduling.

Thus real-time kernels are used for embedded and critical systems that require a very accurate time management and where the failure to comply with the task temporal parameters can result in catastrophic failure of the system and the surrounding environment.

Besides its advantages, software kernels have a major drawback: they share the computational power of the CPU with the application tasks. Due to the inability of most CPU's to provide true task parallel processing, most of the embedded systems are unable to manage periodic and external task activations, run scheduling algorithms and housekeeping functions simultaneously with the application software. Since most of the embedded systems, only have one CPU, some of its processing power, needs to be reserved for kernel execution. This creates a problem not only of performance, but even more important, the introduction of indeterminism. In figure 1(a), we can observe the execution of a full software real-time kernel, where the periodic timer and the interrupt handling routines consumes processing power, hence not available to the applications.



(a) Execution example of a full software kernel implementation.



(b) Execution example of a hardware accelerated kernel.

Fig. 1: Full software versus software/hardware kernel.

Indeterminism is most visible on the external interrupts, due to the inability of knowing when they are going to happen. If in a certain interval of time, several peripherals on the system fire each one an interrupt, the CPU will be overloaded, wasting task processing time and increasing

the probability for the appearance of several problems, such as, task deadline miss (which can be catastrophic), task activation miss, synchronization timing problems and also but not least, decrease of system quality and performance.

By using a hardware supported kernel, it is possible to increase the available processing time by directly controlling the CPU interrupt signal. In figure 1(b), we can observe the kernel execution were the coprocessor controls the processor interrupt signal, managing in a much more efficient way, the available processing time. Also, the wasted processing power will be reduced, since it's the coprocessor that manages all the interrupt requests that comes from the external peripherals. In this way, the CPU is only interrupted when required to preempting the running task and dispatching a higher priority one.

A. Reconfigurable Circuits

When designing and fabricating Systems-on-Chip, the high development and fixed manufacturing costs are attenuated by fabricating millions of chips. However for embedded systems targeting small markets and very specific environments, the requirements can become very unique, so developing and manufacturing custom chips for these specific conditions, becomes completely impossible due to the high costs.

To address these problems programmable or reconfigurable devices, such as FPGAs can be the right answer. Since the manufacturing costs are attenuated by fabricating millions of FPGAs chips, the use of these reconfigurable circuits within embedded systems becomes very attractive. Moreover, since modern systems are often very complex, the probability of containing *bugs* is very high. If the implementation is based on FPGA technology we are able to reconfigure the circuit, allowing to solve the problem without changing the actual hardware or building a new chip.

Despite the potential of reconfigurable circuits, it is not possible to run an FPGA-based systems at the same frequencies that a custom SoC can run. However, this can be mitigated in some applications by a convenient exploration of the parallelism.

B. Paper Organization

Section II shows the motivation for this article, introduces some related work about embedded devices and also presents the OReK real-time kernel. Section III shows the coprocessor details and architecture. Section IV shows how the evaluation was done and the results that were obtained. Then, in the Section V, the conclusions are presented.

II. MOTIVATION AND RELATED WORK

The use of FPGAs in the context of a real-time systems introduce the ability to easily exploit the reconfigurability, in order to provide custom hardware support for the kernel execution. It allows taking the advantage of constructing an application specialized coprocessor that is able to provide the same features of a real-time software kernel, but with the possibility of executing in specialized hardware units in parallel with the application tasks.

Several authors already demonstrated that with hardware support it is possible to reduce the overhead of kernel execution (namely interrupt handling, context switching and task scheduling [1] [2] [3]. Also, in [4] the author demonstrates that the use of a coprocessor, clearly improves the response time of the application. In [1], the acceleration is provided only for the scheduler and in [3], it is only described the interrupt management, despite the author reference to the hardware acceleration of context switching and scheduling.

By using a coprocessor to accelerate the real-time kernel execution, most of the kernel execution overhead can be removed from the processor, remaining only a simplified interrupt servicing primitive for context switching purposes (see Figure 1(b)).

The OReK Kernel

OReK (figure 2), is a full-software, object-oriented real-time kernel for embedded systems, developed in C++. This kernel implements Time Management Services, Task Management and also, binary Semaphores managed by the Resource Stack Policy, for shared resources that require a mutual exclusion access.

The following types of tasks are supported: Soft and Hard real-time Periodic tasks, Soft and Hard real-time Aperiodic/Sporadic tasks and Non Real-time tasks.

The OReK Kernel implements a periodic routine that allows the following actions to be executed: system tick counter update, tasks priority update, periodic tasks activation management according to their temporal parameters, aperiodic tasks activation according to their temporal restrictions and explicit or external interrupt requests, the detection of temporal violations (i.e. deadline miss) and also, the task periodic scheduling.

For the interrupt management, the OReK kernel implements a routine which is executed every time the processor receives an interrupt request. This routine allows the management of aperiodic task activation requests with a minimum of processing time consumption. Since this kernel is a full software implementation, all interrupt requests are managed in a sequential way.

The kernel allows the tasks to be scheduled according to the following schemes: Rate Monotonic (RM), Deadline Monotonic (DM), Earliest Deadline First (EDF) and also First Come - First Served (FIFO).

The OReK kernel porting to PowerPC and Microblaze platforms is described in [5], it was also used in conjunction with the MIPS-based ARPA-MT Real-Time Processor and its tightly coupled ARPA-OSC Real-time kernel hardware accelerator, which resulted in a improved determinism and performance [6] [7]. Although this approach has the disadvantage of not allowing the use of the hardware supported implementation of the OReK kernel in other platforms with other processor architectures without a fully coprocessor redesign.

To allow the use of the hardware accelerated version of the OReK kernel in other platforms, a generic loosely coupled coprocessor architecture needs to be developed, in order to provide a much easier and generic communication infras-

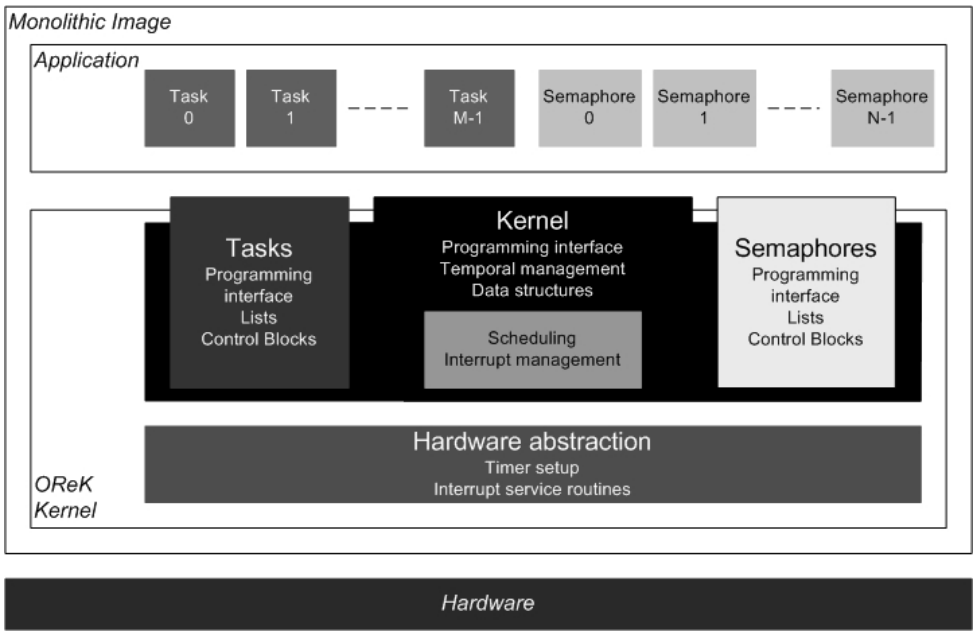


Fig. 2: OReK Kernel Software Architecture

tructure with multiple CPU architectures. This is the aim of this work and will be presented and evaluated in the remaining sections of this paper. The coprocessor described in this paper, provides a complete acceleration of all kernel functions, such as time management, tasks priority updates, tasks scheduling and dispatching, peripheral interrupt requests management, and also semaphore management, allowing a more complete solution.

III. COPROCESSOR ARCHITECTURE

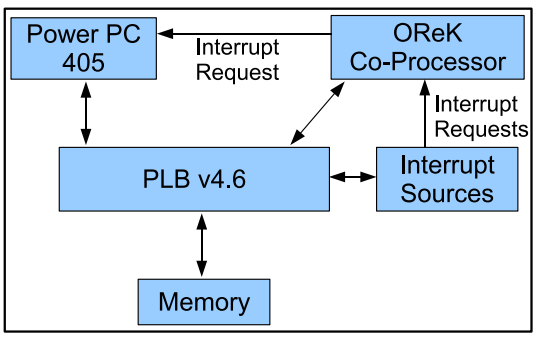


Fig. 3: Overall embedded system architecture with a kernel coprocessor.

In this section, an architecture proposal for a coprocessor is presented (Figure 3 and Figure 4). The coprocessor is constituted mainly by a 64 bit precision timer, an external interrupt manager, task manager, semaphores control block for the resources that require mutual exclusive access, and also, an instruction decoder that is used to access all of it's capabilities. For testing and evaluation purposes, this architecture was implemented in the FPGA Virtex 2 Pro, the coprocessor connects to the system, using the interface for the Processor Local Bus. The output interrupt line is connected directly to the PowerPC 405 CPU, and all of the peripheral interrupt lines, are directly connected to the coprocessor for parallel management.

System Timer

The coprocessor has it's own temporal management through a programmable 64bit precision timer. This timer can be programmed to create a periodic tick with a chosen precision, that is used to control all timing operations such as, task period counting, minimum inter-arrival time analysis and scheduler activation for determining the highest priority task in a certain instant of time.

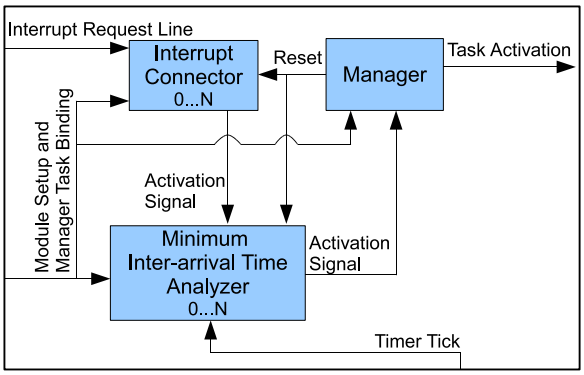


Fig. 5: Interrupt Manager

Interrupt Manager

The Figure 5 shows the internal architecture of the External Interrupt Manager, for the coprocessor.

For each interrupt line there's an Interrupt Connector, that stabilizes the signal for analysis and detection if an interruption is being issued. Depending on the setup, an interrupt can be detected when the signal is in rising edge ($0 \rightarrow 1$), falling edge ($1 \rightarrow 0$) or both. The interrupt detection can

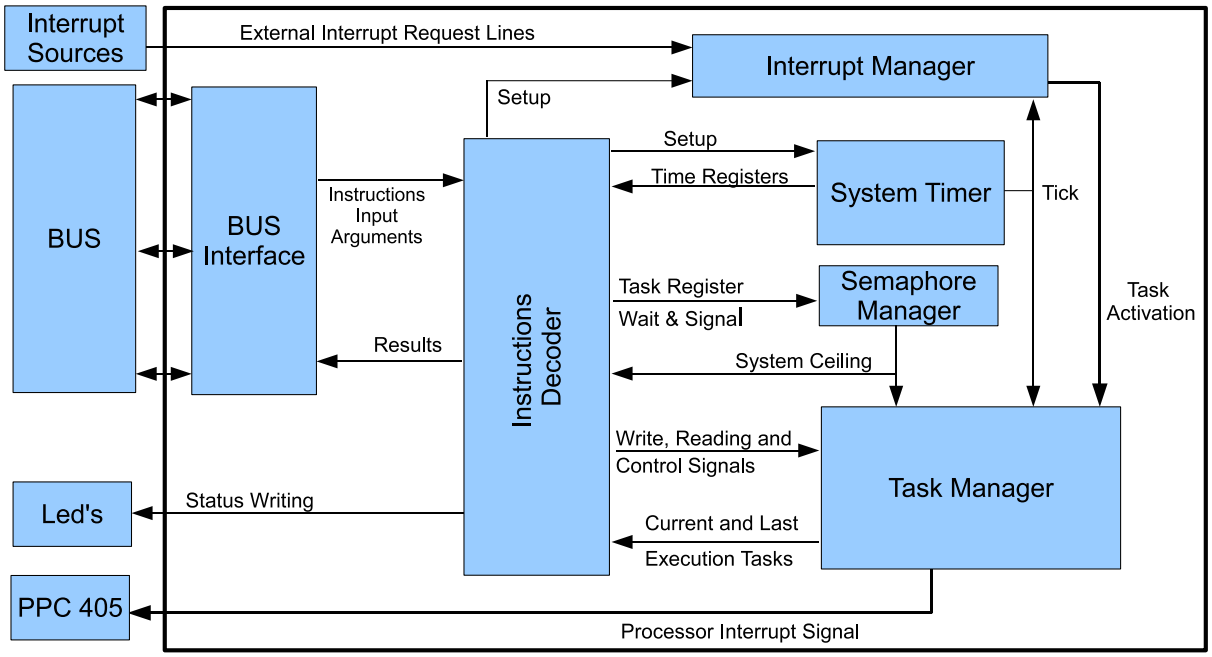


Fig. 4: Coprocessor Architecture - Global View

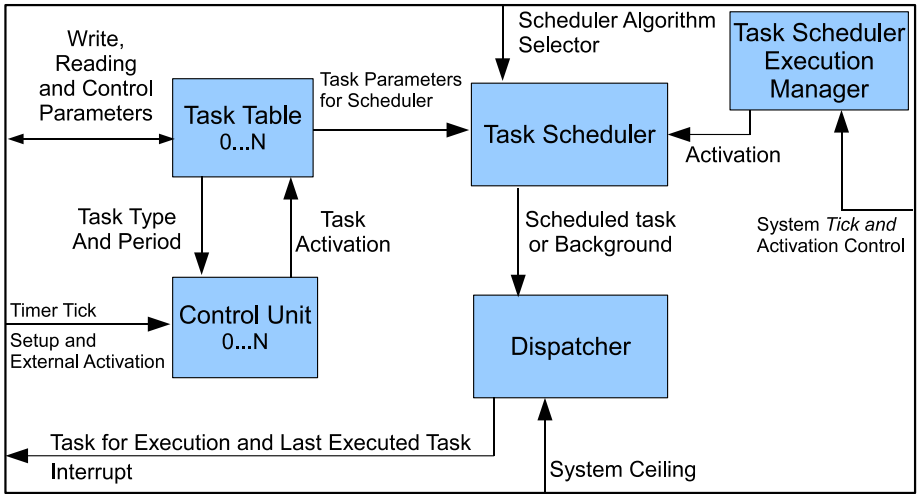


Fig. 6: Task Manager

also be disabled. When an interrupt is detected an activation signal is generated.

The Minimum Inter-Arrival Time Analyzer, which also exist for each interrupt line, receives an activation signal from the IC, and only allows it to pass to the Manager, if the minimum time since the last activation has passed, otherwise, the output will be zero. This block, doesn't delay the signal with clock synchronization, working like a multiplexer.

The Manager, directs the activation signal coming from the MIT Analyzer to the bounded task. The task binding is pre-programmed into the Manager, and also like the MIT Analyzer, the Manager block doesn't delay the interruption signal with clock synchronization, also working like a multiplexer.

Task Manager

The Figure 6 shows the internal architecture of the Task Manager.

The Task Table consists of Task Control Blocks that contain the temporal attributes of each registered task, such as period, relative deadline, absolute deadline, phase and also other special attributes like stop pending signal, deadline missed signal, task type and the task state.

The Control Block, is responsible for the task readiness activation. It generates periodic task activations according to the task period if the task is periodic. If the task is aperiodic/sporadic or non real time, it will receive the activation signals coming from the Interrupt Manager or Instruction Decoder, respectively. Each task is controlled in parallel.

The Task Scheduler contains the Scheduling algorithm (see Section III for algorithm details), which receives from all the tasks, their respective periods, relative deadlines and absolute deadlines, which according to the selected scheduling algorithm, it will output or the task with the highest priority or the background in case there is no task to execute.

The Task Scheduler Execution Manager, is used to control the scheduler activation. In order to prevent an execution request when the algorithm is already under execution, this management unit was implemented.

Finally, we have the dispatcher, which is responsible for the interrupt of the processor for a task preemption, if there is a new task to execute that has a higher priority. If there is no new task with a higher priority, no interrupt is issued.

Semaphore Manager

For resources access synchronization, the coprocessor implements a Semaphore controller, that allows an application to register for a certain semaphore with it's priority level. After that, the task can lock and unlock the semaphore, in order to raise or decrease the System Ceiling, preventing other tasks from taking over the CPU context. The Semaphores use a simplified version of the Stack Resource Policy [8], in order to reduce the amount of logic cells, so there are some restrictions to it's use, such as the order of unlocking must be reversed in relation to the locking order (*Lock A - Lock B - Unlock B - Unlock A*). Also, it's not possible to unregister a single task from a semaphore, it is only possible to reset all of it's registrations.

Scheduler Algorithm

The Figure 7 represents the parallel scheduler algorithm, used to determine the highest priority task in a certain moment of time, in a system that supports eight tasks.

The algorithm execution is very simple. In the first cycle, since the scheduler and the task activation occur in parallel, the scheduler needs to wait for the tasks that were activated during the periodic tick, to change their state to ready.

Then for each task in the system and according to the scheduler policy (Rate Monotonic, Deadline Monotonic or Earliest Deadline First), at the second cycle the scheduler selects a temporal attribute from each task (RM→Period, DM→Relative Deadline and EDF→Absolute Deadline).

If there is no task registered in the task cell, or the task is not ready to be executed, the input will be disabled.

In the third cycle, each task selected attribute is compared 2 by 2 in parallel, reducing the number of tasks to be compared in the next cycle by half, and in the fourth cycle, tasks are again compared 2 by 2, reducing to two tasks and now, in the fifth and final comparison, the algorithm obtains the highest priority task. If there is no task attribute present at the entrance of the Minimum Block, the output will be disabled, and if there is no task available to be executed, the scheduler algorithm will disable it's output, indicating that it's the background that should occupy the CPU context.

The complexity of the algorithm is $O(logN)$ and by using the formula

$$Cycles = 2 + log_2 N$$

we can determine the exact number of cycles that the algorithm will require to obtain a result, in order to the maximum number of tasks supported by the coprocessor(N). This guarantees, that for a certain number of tasks, the algorithm will always takes the same number of cycles to obtain a result, making it truly deterministic.

IV. EVALUATION AND RESULTS

The FPGA that was used for test and evaluation, was a Virtex 2 Pro, which internally has a PowerPC 405 Processor. The operation frequency for both processor and FPGA were 300Mhz and 100Mhz respectively. The Instruction Segment were placed in a special memory, directly connected to the PowerPC and the Data Segment along with the Heap and Stack, were placed at the System Memory. In order to connect the System Memory and the Coprocessor, a Processor Logic Bus (v4.6) was used (see Figure 3).

Since the Virtex 2 Pro has few logic cells, it was only possible to synthesize the coprocessor in order to support a total max of 16 tasks along with 4 semaphores and 5 external interrupt lines. But be aware, this is only a FPGA resource limitation, not an architectural one.

In [5], the author already evaluated the OReK kernel in full software implementation, so for this article, the evaluation will cover only the kernel execution with the coprocessor support. Also, the use of the PLB introduces a high delay when accessing the system memory or the coprocessor registers, since it takes 138ns just to read one 32bit register, and 201ns to write to a register, so this aspect should be taken in consideration when observing the system primitives execution timings [4].

To measure the time that each kernel primitive requires to execute, the processor timer was used. The PowerPC 405 has a 64bit internal clock that starts running right at system boot, and by accessing their registers, it is possible to obtain a time value with a precision of 3,3ns.

There are some special primitives, which require special explanation. The Context Change routine is executed every time the coprocessor generates an interrupt. Take in consideration that the measured time in the table does not contain the cycles that are required by the scheduler to obtain the highest priority ready-to-execute task. On the other hand, the Task termination routine, already contains the cycles required by the scheduler, since that for every ending task, the coprocessor is ordered to schedule a new task to replace the processor context.

For the semaphores primitives, the Enable Semaphore primitive is not applicable when using a coprocessor, simply because the semaphores are always enabled. This primitive is only valid in the full software version of the OReK kernel.

V. CONCLUSIONS

This paper presented the architecture of a hardware coprocessor for the OReK Real-Time Executive. With the use of this co-processor, periodic and aperiodic tasks activation determinism is improved and all tasks activation are done in parallel. This means that it is now possible to activate all tasks simultaneously, allowing a *by-the-clock* accuracy, completely removing the indeterminism associated to an unpredictable number of tasks to be set to a ready status and, unpredictable number of consecutive external interrupt requests, since now all external interrupts are fully controlled and monitored by the coprocessor.

Also, the use of the coprocessor reduces substantially the time necessary for a tasks context switch and task termina-

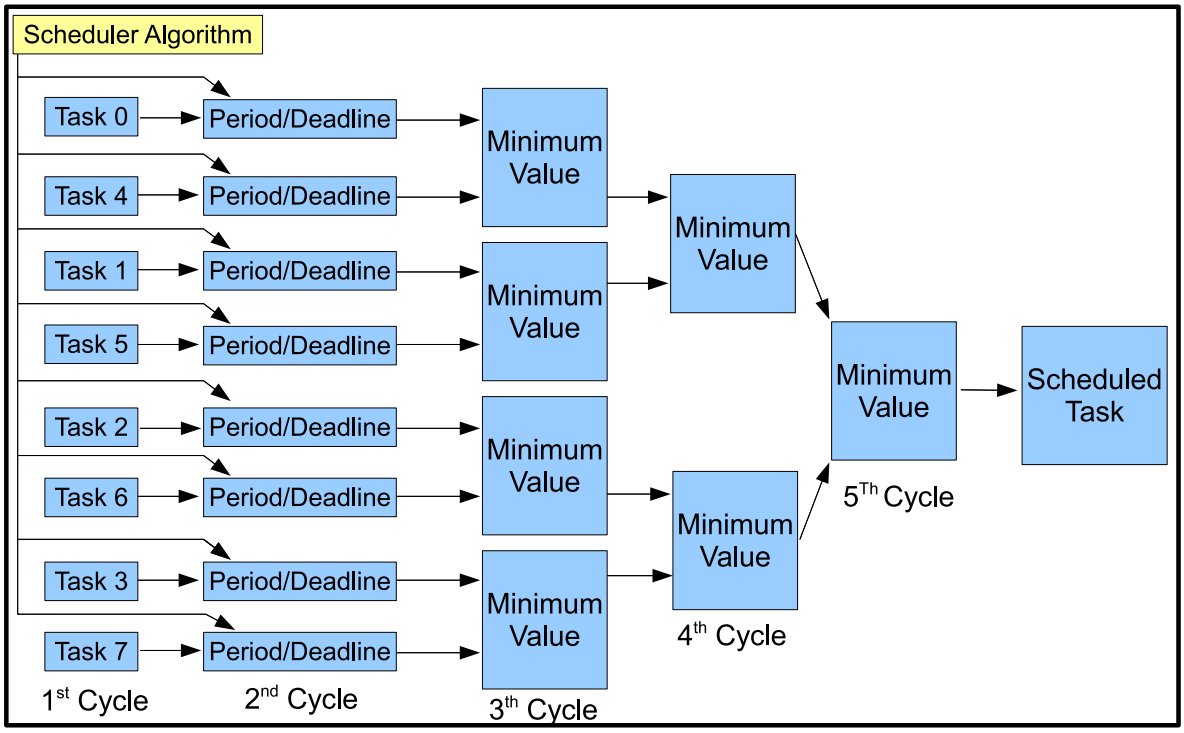


Fig. 7: Scheduler Diagram

Parameter Evaluated		Implementation OReK-CP Non Cached	Implementation OReK-CP Cached	# Tasks
Kernel Special Execution Routines				
Task Context Change	BG→Task	7,870μs	1,323μs	Any
	Task→Task	7,893μs	1,343μs	
Task Termination	Task→BG	6,722μs	2,237μs	4
	Task→Task	6,959μs	2,257μs	
	Task→BG	6,722μs	2,237μs	8
	Task→Task	6,959μs	2,257μs	
	Task→BG	6,791μs	2,296μs	16
	Task→Task	7,029μs	2,316μs	
Kernel Primitives				
Kernel Initialization		30,808μs	18,810μs	4
		32,234μs	19,641μs	8
		34,838μs	21,423μs	16
Kernel Shutdown		27,297μs	13,305μs	4
		40,286μs	19,503μs	8
		65,567μs	30,765μs	16
Start Kernel Execution		1,227μs	0,752μs	Any
Stop Kernel Execution		0,475μs	0,376μs	
Disable Interrupts		0,227μs	0,221μs	
Enable Interrupts		0,227μs	0,221μs	
Get Tick Count		1,283μs	0,531μs	
Disable Preemption		0,498μs	0,353μs	
Enable Preemption		0,498μs	0,372μs	
Create Non-RT Task		10,470μs	5,187μs	
Create Soft-Periodic Task		10,530μs	5,128μs	
Create Soft-Aperiodic Task		12,668μs	7,999μs	
Create Hard-Periodic Task		10,589μs	6,157μs	
Create Hard-Aperiodic Task		12,688μs	8,038μs	
Destroy Task		6,553μs	3,742μs	
Start Task		11,206μs	5,563μs	
Stop Task		3,286μs	1,940μs	
Activate Task		2,633μs	1,564μs	
Get Task State		3,069μs	1,980μs	

TABLE I

KERNEL TIMING EVALUATION - TASK PRIMITIVES

Parameter Evaluated	Implementation OReK-CP Non-Cached	Implementation OReK-CP Cached	# Semaphores
Create Semaphore	7,266μs	3,722μs	Any
Destroy Semaphore	6,296μs	3,362μs	
Register Task On Semaphore	4,039μs	2,296μs	
Enable Semaphore	Not Applicable		
Wait on Semaphore	3,326μs	1,821μs	Any
Signal Semaphore	3,148μs	1,603μs	

TABLE II

KERNEL TIMING EVALUATION - SEMAPHORES PRIMITIVES

Coprocessor Internal Routines	Cycles	Time
External Task Activation	5	50ns
Periodic Task Activation	2	20ns

TABLE III

COPROCESSOR EXECUTION TIMINGS

Parameters	Value
Number of Tasks	2,4,8,16,32,64,128,256
Number of Interrupt Lines	0-255
Number of Semaphores	0-255

TABLE IV

COPROCESSOR SYNTHESIS PARAMETERS

tion routines execution, leaving more processing time for the tasks and improving the efficiency of the system. It also improves the determinism because, now it is the dispatcher that orders an interrupt of the CPU execution for a context switch, the scheduler is actually executed inside the coprocessor, in parallel with the task processing, removing the

Number of Tasks	Flip-Flops	LUT's	Slices
4	3243	5943	3302
8	4673	9623	5137
16	7529	16805	8945
32	12303	30765	16307

TABLE V
COPROCESSOR RESOURCES USAGE FOR 5 INTERRUPTION LINES AND 4 SEMAPHORES

dependency from the number of tasks registered in the system, leaving with a code execution that is very deterministic and very fast.

Unfortunately, due to high latency generated by the PLB and PLB interface [4], the kernel execution primitives became very slow when compared to the full software OReK kernel version [5], although if better ways are used to interconnect to the coprocessor instruction decoder and internal registers (i.e. direct connection from the CPU to the coprocessor), the timings can be greatly improved. The use of the PLB was imposed by the testing platform and not by the coprocessor architecture.

To conclude this article, the coprocessor can manage periodic and aperiodic tasks scheduling and activation, impose time limits for the external interrupt requests and helps to synchronize shared resources through the use of internal semaphores. With these capabilities, the coprocessor greatly improves not only performance but more important, the determinism. So, it is now possible to simulate the system behavior, due to the removal of software kernel dependencies on unpredictable factors, such as, external interrupt requests overload, number of tasks registered in the system, number of tasks to be activated in a certain moment of time and also but not least, task synchronization factors related to the system ceiling.

REFERENCES

[1] Youngchul Cho, Sungjoo Yoo, Kiyoung Choi, N.-E. Zergainoh, and Ahmed Amine Jerraya, "Scheduler implementation in MP SoC design", in *ASP-DAC'05 - The Asia and South Pacific Design Automation Conference*, Shanghai - China, Jan. 2005, pp. 151–156, IEEE.

[2] S. Nordstrom and L. Asplund, "Configurable hardware/software support for single processor real-time kernels", Nov. 2007, pp. 1–4.

[3] Moonvin Song, Sang Hoon Hong, and Yunmo Chung, "Reducing the overhead of real-time operating system through reconfigurable hardware", *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pp. 311–316, Aug. 2007.

[4] Johan Furunäs, "Benchmarking of a real-time system that utilizes a booster", 2000.

[5] N. Silva, A. Oliveira, R. Santos, and L. Almeida, "The orek real-time micro kernel for fpga-based systems-on-chip", in *Embedded Systems for Real-Time Multimedia, 2008. ESTImedia 2008. IEEE/ACM/IFIP Workshop on*, Oct. 2008, pp. 75–80.

[6] António B. Ferrari Arnaldo S. R. Oliveira, Luís Almeida, "A specialized and predictable processor for real-time systems", in *A Specialized and Predictable Processor for Real-time Systems, 2007.Workshop on Application Specific Processors*, Oct. 2007.

[7] António B. Ferrari Arnaldo S. R. Oliveira, Valeri Skliarov, *Especialização e Síntese de Processadores para Aplicação em Sistemas de Tempo-Real*, PhD thesis, 2007.

[8] T. P. Baker, "Stack-based scheduling for realtime processes", *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, 1991.