An Embedded, Generic and Multiprocessor Hardware Operating System

Fabrice Muller
University of Nice Sophia-Antipolis,
LEAT/CNRS
250 Rue Albert Einstein, Bât. 4, Les Lucioles 1
06560 VALBONNE - France
Fabrice.Muller@unice.fr

Farooq Muhammad
University of Nice Sophia-Antipolis,
LEAT/CNRS
250 Rue Albert Einstein, Bât. 4, Les Lucioles 1
06560 VALBONNE - France
Farooq.Muhammad@unice.fr

Abstract

This paper presents a hardware real time operating system (HW-RTOS) for multiprocessors. A Virtual Platform (SystemC) and Prototyping Platform (FPGA board) have been designed to be generic, modular and to support multiprocessor architectures and the HwRTOS. Thus we can customize functionalities and scope of platforms according to the needs of an application. The HwRTOS offers classical OS services, new services to improve multiprocessor management (migration of tasks) and possibilities of scheduling analysis. Indeed, these platforms can be also used to develop new scheduling algorithms to improve the load balancing but also to take into account others parameters such as low power parameters.

1 Introduction

With the increasing complexity of applications, Multiprocessor System on Chip (MPSoC) becomes an important choice for implementation. But, MPSoC solutions for real time systems require an efficient Real Time Operating System (RTOS) to manage the resources and to guarantee the real time constraints. However, the verification of the application running on the MPSoC platform grows to be too complex due to the increase of number of processors, the application complexity, the bus interconnections and the management of resources by one or more OS. Indeed, management of resources and scheduling of applications distributed on processors are also complex problems.

We propose an approach which includes a Virtual Platform and a Prototyping Platform composed of a number of heterogeneous processors, communication busses and a generic HwRTOS (Hardware Real Time Operating System) to manage the MPSoC platform. We developed a generic RTOS in hardware which could be used directly on the Virtual Platform (SystemC) and Prototyping Platform. It is also possible to make decisions about scheduling techniques (Rate Monotonic, EDF) and scheduling taxonomy (Global, Local, Hybrid), and adapt these choices to the varying needs of application at run time.

Indeed, this HwRTOS is quite generic. We can easily configure it according to the user needs and application demands, just a few variables need to be specified like the number of processors, the number of tasks, the synchronization methodology and the selection of OS services. Moreover, one can choose any predefined scheduling algorithm or user-defined scheduling written in hardware language for the Prototyping/Virtual Platform or SystemC language for the Virtual Platform only. Thus, the HwRTOS is scalable in relation to the application.

This paper is organized as follows: Section 2 presents some existing software and hardware RTOS. Section 3 introduces the Hardware RTOS. Section 4 details the hardware part; a new way to develop scheduling algorithms; the description of the hardware modules; Section 5 presents some results about implementation of the HwRTOS. Section 6 illustrates the new proposed flow through two types of application. Conclusions are available in section 7.

2 State of Art

We propose a Multiprocessor Hardware RTOS to perform applications running on processors. The HDL code of the HwRTOS runs on virtual platform and prototyping platform. It limits the risk of wrong behaviors of the application on different levels of abstraction. Indeed, the behavior of Hardware RTOS is the same on the virtual platform and the prototyping platform. We propose to glance through the management of applications on MPSoC to well-define the context of work.

One solution often used for the management of applications is an RTOS. Indeed, some RTOS include specific services to carry out memory or input/output management, but all these functionalities are implemented in software and are supported by the processor. For example, VxWorks or RTLinux are very complete. Thus our goal is not to rival with these RTOS, but to define an efficient hardware implementation and evaluate its benefits.

The idea of a hardware Operating System that moves scheduling and inter-process communication from software to hardware has been addressed in some previous works. The main idea is to move the RTOS functionalities that consume more CPU power into hardware in order to benefit from hardware acceleration.

SiliconOS [3] is a full-fledged operating system in which the majority of the µITRON functionality is implemented on a coprocessor called Silicon TRON. SiliconOS does not support multiprocessor architectures. Moreover, services like memory and timer management

that consumes processor time are still implemented in software. The resulting software kernel is one third the size of the original software kernel.

The δ SoC Codesign Framework [4] is built around the Atlanta kernel [1] and allows a more fine-grained partitioning with respect to [3]. This kernel provides key RTOS features including multitasking capabilities, event-driven and priority-based pre-emptive scheduling, inter-task communication and synchronization, but it does not permit to change the scheduling at run time. HOPES [2] is a RTOS-like system that allows run time partitioning and allocation of reconfigurable FPGAs. It supports both pre-emptive and non pre-emptive scheduling methods but does not provide multiprocessor scheduling and semaphore services. FASTCHART [3] is a real time kernel fully implemented in hardware. Key features of this kernel are: priority scheduling, synchronization primitives and interrupt handling but the last version (Sierra 16) does not support multiprocessor architectures. Despite this amount of previous work and initial industrial attempts like [3], at present commercial RTOS does not offer generally:

- Multiprocessor support with the possibility of dynamic load balancing (global and local scheduling),
- Ability for designers to modify policies at run time. Users can decide only offline which algorithms to implement in hardware and cannot adapt to user needs at run time.

We propose an embedded and generic hardware multiprocessor RTOS where the user can select statically the different scheduling algorithms. It can also change the scheduling policies at run time as well. The effort in this work focused on not only implementing RTOS in hardware, but also on providing flexibility to the user for changing decision-algorithms at run time thanks to VHDL or SystemC algorithms within the HwRTOS.

3 New Hardware RTOS

The HwRTOS is the heart of the SoPC platform. It can manage complex applications on a homo/heterogeneous multiprocessor architectures. There are some generic parameters (number of processors, number of tasks …) to adapt the HwRTOS for a target application.

It supports most of OS services implemented in Hardware as semaphores, message queues, kernel services and debug services. It is also possible to add easily new hardware services by a user. In the same way, new scheduling algorithms could be added in the hardware algorithm module located in the HwRTOS.

The last point concerns the debug. Indeed, the HwRTOS proposes a build-in hardware *Debug* module to capture or to spy all the OS events. The *Debug* module is configurable by software (task scanning, triggering, command filtering). The trace could be sent to the Trace Analyzer through a UDP Ethernet to extract performances of the architecture and the application (overhead, execution time of tasks, response time, scheduling).

3.1 Software Layers of the HwRTOS

One challenge is to have the same software legacy code of the application as explained in Figure 1. The application is composed of local tasks, the shared data, the code of global tasks which can migrate from one processor to another and the generic parameters to keep the coherency between the HDL code and the software part. The HwRTOS middleware is split in two parts. The upper layer is the code of the OS services which access the registers of the HwRTOS to perform the hardware part of the service.

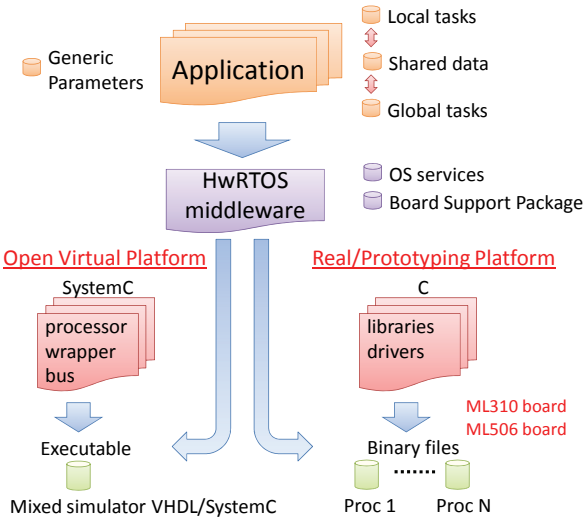


Figure 1 : Software organisation of the RTOS.

The Prototyping Platform has a target like ML310 board (VirtexII Pro) or ML506 board (Virtex 5).

3.1.1 Behavior of a Service

Each service runs in similar fashion. To illustrate this, let's consider the example (Figure 2) of the task delay service. The first few steps are executed in a critical section. The first step switches on critical section. Step 2 configures the *wait* field of the current TCB. Step 3 sends command to the hardware module. We write the task id, the delay value and the command that triggers the execution of the hardware part of service.

```
#define taskDelay(id, delay) \
        rtos_taskDelay(HWRTOS_PROCESSOR_NUMBER, id, delay)

status_type rtos_taskDelay(int procNumber, task_id_type id, int delay)
{
    1  setDisableInt();
    2  sw_kernel.tcb->wait = bTRUE;
       WriteReg32(procNumber, PARAM1_OFFSET, id);
    3  WriteReg32(procNumber, PARAM2_OFFSET, delay);
       WriteReg32(procNumber, COMMAND_OFFSET, TASK_DELAY_CMD);
    4  setEnableInt();
    5  my_tcb = sw_kernel.tcb;
       while (my_tcb->wait == bTRUE);
    6  status = my_tcb->status;
       return getErrorStatus(status);
}
```

critical section

Figure 2 : Behavior of Task Delay service.

Step 4 allows exiting of the critical section. Step 5 blocks the task while the *wait* field or the *ack_event* field is not valid. The interrupt handler validates the *wait* variable and releases the Step 5. In general case, it is not mandatory that the step 6 of the current task is executed immediately. Actually it depends on the new elected task decided by HwRTOS.

3.1.2 Standard OS Services

Each hardware module manages services (semaphore, message queue, kernel and scheduler). The names of the services are inspired from those defined in VxWorks, in a way to simplify the understanding and use of the HwRTOS. The different services can be used very easily in all processors. The way of coding of an application is identical to application coding in case of a mono processor. HwRTOS is a native multiprocessor RTOS with built-in multiprocessor services. For example, it manages synchronizations (semaphores) or message queues between tasks allocated onto different processors.

3.1.3 New Custom OS Services

The HwRTOS is a multiprocessor RTOS. Thus, it proposes new interesting OS services to manage efficiently several processors. The HwRTOS supports two types of tasks: the local tasks and global tasks. A local task always runs on the specific processor defined at the compilation step. Global tasks can be preempted at any time and can migrate between processors. This migration of task is supported by Prototyping platform. The migration is not considered as a service but a native characteristic inside the HwRTOS.

Firstly, we propose new scheduling selection services to modify on-line scheduling policy globally or locally. The decision of modification is done by the application or a hardware module.

Secondly, the HwRTOS is also used to develop new scheduling algorithms. Most of these algorithms use periodic tasks. The semaphore service could be used but it is not enough accurate due to the overhead. Moreover, it is more complex to put in its place.

3.1.4 Debug Service

The debug service allows capturing a lot of encapsulated events coded on FIFOs. We propose primitives to drive the debug service:

- *Filtering primitives*: They allow selecting debug events: tasks, semaphores and message queues identified by an ID. The others debug events are not added in the trace. This solution reduces the size of the trace.
- *Triggering primitives*: It is possible to start the capture of the trace when you want and by processor.
- *Trace primitives*: The trace is stored in FIFOs (one per processor). These primitives access to the FIFO in order to read trace.

When a FIFO is almost full, a processor has to read the FIFO. The Prototyping Platform can store trace on memory or flash system or to send the trace to the Performance Analyzer by RS232 protocol and UDP protocol (Ethernet).

4 Hardware Part of the HwRTOS

In a way to decrease the processor load, resulting from RTOS execution and its multiprocessor support, an idea is to move functionalities of the RTOS to hardware, as seen in previous part. It is possible using current SoC or MPSoC that are more flexible. The first work has been to identify and cut out the RTOS into different modules that could be implemented in hardware. Another important characteristic is the multiprocessor aspect. It is necessary not only to modify easily and statically the number of processors, but also to change number of tasks per processor, the number of semaphores and so on. All hardware RTOS mentioned before does not support this flexibility, except Atlanta [1].

The first part of the study focused on the functionalities of the HwRTOS that are essential. Indeed, 90-95% of all RTOS functionalities are limited to the services [4]: create task, get and release memory buffer, send a message, receive a message with waiting or with a time-limit on waiting. On the HwRTOS, we have implemented all services except memory management. Moreover, in this approach, all these services are implemented in hardware. The software part is reduced to the minimum.

4.1 Specifications

Considering the multiprocessor architecture of Figure 3, we propose to allocate task execution on set of processors and to use HwRTOS services to schedule tasks and to synchronize the communications between tasks by semaphores or message queues. The goal is to run computational intensive parts of the HwRTOS in hardware to improve the performances and to control all processors at any time. We have just kept a minimal software layer to responds quickly to the commands of the hardware part.

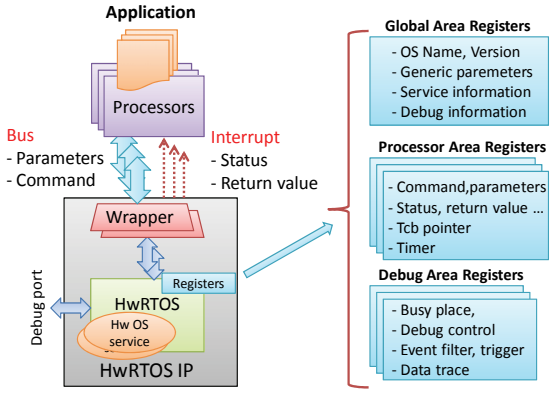


Figure 3 : Interaction with Environment.

The software layer sends parameters and a command associated to an OS service, towards the hardware part

that triggers the execution of hardware OS services. The HwRTOS core can be connected to any bus. The designer has to create a wrapper to adapt the protocol of the HwRTOS generic bus. The capability of multi-port is possible to separate the debug port and the application ports. That does not cause any disturbance in behavior of the application in the Virtual Platform and Prototyping Platform.

Each processor is assigned a reserved register area for OS services and debug. A processor can write parameters, write commands, read status, read return values, configure debug and recover debug trace. Each processor can also access on a global area to obtain general information about the main characteristics of HwRTOS like number of processor, number of task, implemented services and so on. As soon as a command is completed by the hardware part, two kinds of events or interrupts can occur:

- A "command event" informs the processor that a return value is available or that the service is completed with or without errors (by reading the status register).
- A "schedule event" informs the processor that a schedule has to be done by the algorithm scheduling module. When this schedule event occurs, information stored in registers indicates either if the system keeps on executing the same task or if it has to switch to another one. In this case, we have a migration flag, the number of the previous processor and the TCB (Task Control Block) of the new task.

Moreover, a task on a processor N may require some modifications to appear (if possible) in the scheduling of tasks on another processor M. For example, task T_i on processor N may suspend the execution of task T_j on processor *M*. On the user side, a suspend service, called on one processor, can suspend a task running either on same processor or on any remote processor. All other services like *rtosSchedulingSet*, *taskLock*, *taskDelay* and others have the same behavior.

4.2 Generic Parameters of the HwRTOS

The power of the HwRTOS is the configuration by static generic parameters. These parameters can be defined by the user. All generic parameters are programmable before synthesis. Indeed, the objective is to adapt the parameters to the application. We propose main generic parameters: the technology parameter (VirtexIIPro, Virtex4 and Virtex5), the number of processors (1 to N), the number of task per processor (2 to M), the number of semaphore, the capability of global scheduling and finally the SystemC algorithms option (only for the Virtual Platform).

We also have advanced generic parameters to change the scheduling algorithm, the priority of treatment of each service, width of all data in the HwRTOS and so on. We have about more than forty advanced generic parameters to adjust more precisely the HwRTOS for a domain of application.

4.3 Hardware Modules

The hardware part is composed of eight main modules (Figure 4). The *Interface* module handles the order of request to all processors. The others modules are the *Delay* module, the *Semaphore* module, the *Message Queue* module, the *Sensor* module and the *Tick* module. The *Scheduler* module is more complex and has been refined by a *Scheduler Controller* module and an *Algorithms* module. Actually, the *Scheduler* module manages the scheduling algorithms and makes decisions about running tasks for each processor. Lastly, the *Debug* Manager captures relevant events in the HwRTOS to be sent to the Performance Analyzer Tool. Each module communicates inside the HwRTOS core through three kinds of bus:

- The *Interface* bus has one master (the *Interface* module) and several slaves (All others service module except the *Scheduler* module),
- The *Next Task* bus is bidirectional, point to point bus. It sends to the *Interface* module (Target) the scheduling information. The *Interface* module provides information of used processor and the current running task on each processor.
- The *Scheduler* bus has multiple masters. Each module connected to this bus can be a master.

The advantage of this connectivity is the "plug & play" approach. Thanks to this modularity, it is possible to add easily new hardware services connected to the *Scheduler Bus* and the *Interface Bus*. We now describe each module.

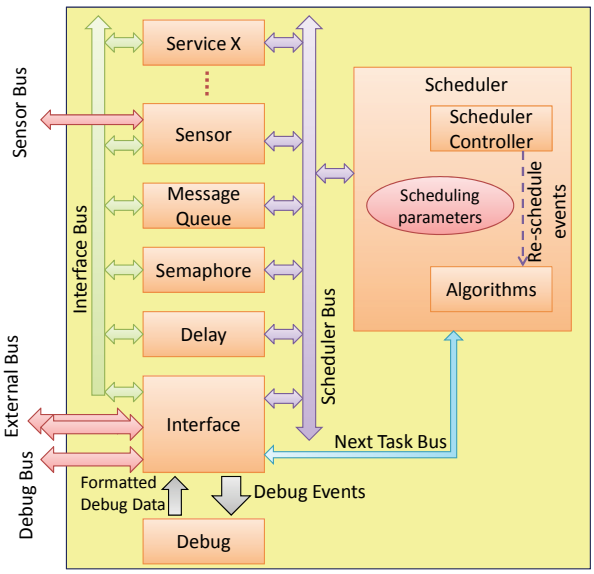


Figure 4 : Functional View of HwRTOS core.

4.3.1 Interface Module

The hardware *Interface* module (Figure 5) is designed to ease the accesses of registers through a generic parallel bus protocol which is portable with any standard bus (PLB, OPB …). We have three main areas. The global area which provides general information about the

HwRTOS, the processor area where each processor has a dedicated area and the debug area which is dedicated for each processor to deal with the debug management

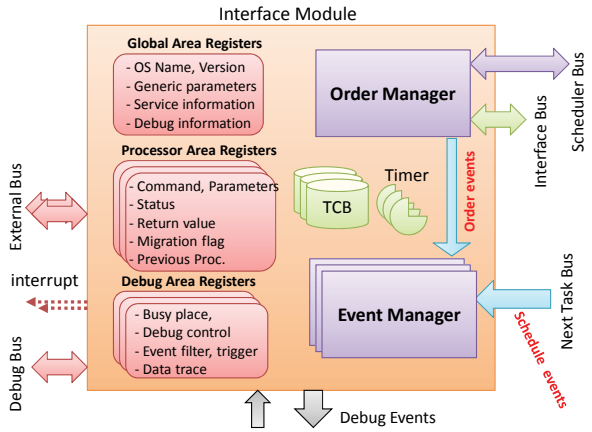


Figure 5 : Description of the Interface Module.

The *Order Manager* interacts directly with the *Scheduler* module through the *Scheduler Bus* for some services like the creation and deletion of task, the critical section (lock, unlock), the suspend service and resume services. It also communicates with each hardware service through the *Interface Bus* to manage Semaphore service, Message Queues service and so on. Finally, the *Order* manager is a big finite state machine that dispatches commands to all other modules through one of two buses.

There are many instances of the *Event* manager, *order events* and *schedule events* depending on the number of processors. Each *Event* manager is assigned to a processor and it manages these own events to transform them in interrupt signal sending to concerned processor.

We can notice that the interface has one port or multi-port capability to connect to processors. The interface supports either one port to connect all processors, or to connect one bus per processor, or another combination.

4.3.2 Semaphore Module

The *Semaphore* module is another important module. Usually, a semaphore helps to synchronize tasks running on the same processor, but the semaphore implemented in the HwRTOS provides a synchronization mechanism for tasks running on different processors. When a semaphore is created, the *Semaphore* module assigns it a unique ID, a value (binary or counter), and a task-waiting list. The user also specifies the type of semaphore: binary, counting or mutual-exclusion (mutex).

4.3.3 Message Queue Module

The *Message Queue* module can also synchronize tasks by sending messages between different processors. The module stores the 32 bits pointers of the data structure. The shared messages have to be stored in an external shared memory when the tasks mapped onto different processors want to communicate among themselves. The behavior of the Receive primitive is the same as the Give primitive of the *Semaphore* service.

4.3.4 Scheduler Module

The *Scheduler* module is the heart of the HwRTOS and includes scheduling parameters, the triggering of new schedule and the management of the scheduling algorithms. It makes scheduling decisions depending on the states of the task set and others scheduling parameters like priority, deadline and of course the scheduling policy. It also helps the user to switch dynamically between global and local scheduling policies for multiprocessor systems, and allows keeping trace of different states of a task. A task has different states: dead, ready, running or blocked. The blocked state of a task can be refined by three states: delayed state for the task delay service, suspended state for the suspend/resume services and pended state for the semaphore service or the message queue service. This information (states, scheduling parameters, scheduling policies) are stored in registers of ram blocks. This strategy of keeping task states at a global level helps switching between local and global scheduling. There is one central *Scheduler Controller* module that triggers the *Algorithm* module to elect the next tasks.

4.3.5 Debug Module

The *Debug* module allows debugging the applications which are managed by the HwRTOS. The principle is to capture information (four types of events and associated data) on the HwRTOS coming from the interface (Figure 5), to format the information for forming a trace and store them into FIFOs (one per processor). These FIFOs are read by any processors through the processor ports or the debug port in order to transmit to the Performance Analyzer by Ethernet communication or Files.

4.4 New approach for scheduling

The scheduling algorithm is normally coded in software. Our approach is to describe the algorithm in HDL or SystemC for Virtual Platform only. To achieve this, we refine the *Scheduler* module by two sub-modules as shown in Figure 6: the *Scheduler Controller* module that arbitrates the *Scheduler* bus and manages the *Algorithms* module according to incoming requests. The behavior of the *Scheduler Controller* module can be summarized in three stages:

- We have a module requests which are an update of the parameters by the Service modules. For example, the *Delay* module wants to change the state of a task from delayed to ready. Moreover, the Algorithms module is able to request a re-schedule through the *Scheduler Controller* module. The *Algorithms* module is allowed to re-schedule itself.
- When all transactions between external modules (Interface module, Delay module …) and *the Scheduler Controller* module are finished, a set of re-schedule events are sent to the *Algorithms* module. One re-schedule event is assigned on each processor. Thus, the number of events sent depends on the module requests or algorithm request.

- When the computation of the *Algorithms* module is completed, the elected tasks are sent to the *Interface* module through the *next task* bus.

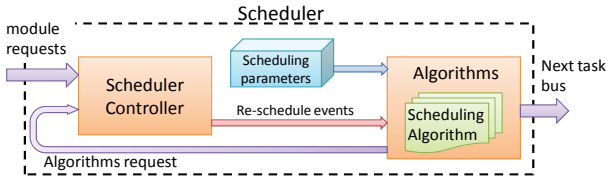


Figure 6 : New Approach for Scheduling.

This structure based on the *Scheduler* module associated with the *Algorithms* module allows a designer to develop easily new HDL/SystemC algorithms for multiprocessor management.

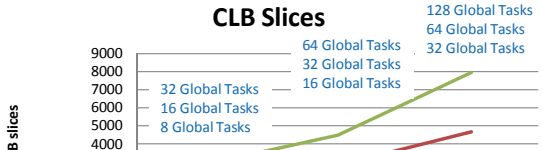
4.5 New Approach for Debugging

The debug of the behavior of the application is possible by the hardware *Debug* module inside the HwRTOS. The *Debug* module records events of the *Interface* module. It allows recording events coming from the *Interface* module for the Virtual Platform or Prototyping platform. It is an important advantage because the comparison of a virtual trace and prototyping trace becomes possible.

We have developed a Performance Analyzer tool in Java language which extracts main characteristics of the application: overhead, number of context switch, number of migration, processor load and the OS services called. The Performance Analyzer makes easy the evaluation of the behavior and the performances of applications running on the Prototyping platform.

5 Implementation of the HwRTOS

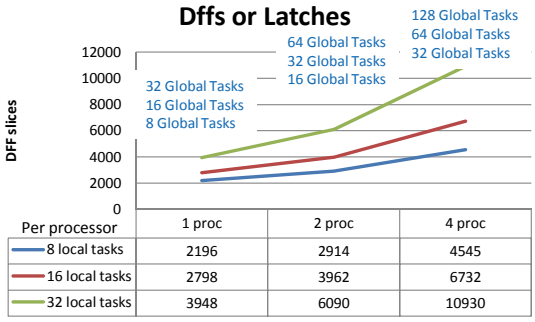
The HwRTOS implementation has been tested in VirtexII Pro, Virtex4 and Virtex5 technology. The Figure 8 and Figure 8 sum up the resources for configurations of the HwRTOS for Virtex5 target.



Per processor	1 proc	2 proc	4 proc
8 local tasks	1507	2035	3037
16 local tasks	1998	2816	4668
32 local tasks	2853	4478	7958

Figure 7 : CLB Utilizations on Virtex5 SX50.

We have fixed 8 semaphores, 8 message queues and no debug service. We also include the local scheduling mode (by priority and parallel computation) and mixed scheduling mode (by priority and sequential computation). To obtain the total number of schedulable task for a configuration, you must add the local tasks and global tasks. For example, for 4 processors and 8 local tasks, the HwRTOS can schedule till 48 tasks.



Per processor	1 proc	2 proc	4 proc
8 local tasks	2196	2914	4545
16 local tasks	2798	3962	6732
32 local tasks	3948	6090	10930

Figure 8 : DFFs Utilizations on Virtex5 SX50.

6 Case Study

We propose two examples in order to illustrate the new flow. The first deals with a classical application running on a multiprocessor architecture. The second one is a typical application to study new scheduling algorithms.

6.1 First Example

We propose a simple example (Figure 9.) which uses the most of services. The application is composed of six tasks partitioned on two processors. The *GenPt1* task sends two messages through a message queue *MsgQ_1* corresponding to randomize values X and Y of a table stored in a shared memory. The table is protected by mutex to avoid the problem of coherency. The *WrSqrPt1* task waits the messages and increments the case (X,Y) on the table. The *GenPt2* task and *WrSqrPt2* task have the same behavior as the *GenPt1* task and *WrSqrPt1* task except the period. Then, the *CheckSqr* task waits two semaphores, next checks and saturates if necessary the square of the table when the number of the square is superior to N. After this scanning of the table, a semaphore is sent to the *Display* task to display the table on the console.

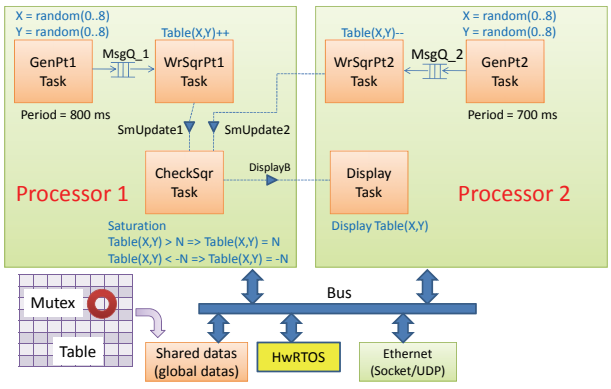


Figure 9 : Application mapped onto the platform.

6.1.1 Application Description

To illustrate the code of the application, the Figure 10 represents the code of the *WrSqrPt1* task. The task waits sequentially the X value and the Y value to form the point

by using the blocking receive primitive of the Message Queue service. The second stage consists of incrementing the table when the task takes the mutex. Finally, the task informs about the update of the table the *Display* task by a give primitive of the Semaphore service.

```
void WrSqrPt1_Task()
{
    int X,Y;
    StartPoint();
    while (1)
    {
        /* Wait X Value */                                   <- Wait point (X,Y)
        msgQ_Receive(msgQP1_ID, WrSqrPt1_ID, &X, MSGQ_BLOCKING);
        /* Wait Y Value */
        msgQ_Receive(msgQP1_ID,WrSqrPt1_ID, &Y, MSGQ_BLOCKING);

        sem_Take(semMutexTable_ID, WrSqrPt1_ID);         <- Mutex Section (10 us)

        table[Y][X] = table[Y][X] + 1;
        CodeDuration(1000);

        sem_Give(semMutexTable_ID, WrSqrPt1_ID);

        /* Send Semaphore To CheckSqr task */                <- Send Update
        sem_Give(semC_updateP1_ID, WrSqrPt1_ID);
    }
}
```

Figure 10 : Code of the WrSqrPt1 Task.

6.1.2 Evaluation of Performances

Firstly, the application has been tested on the Virtual Platform. We add execution time inside all tasks with the *CodeDuration* primitive as shown in *WrSqrPt1* task. In this example (Figure 11), the processors are not loaded especially the processor 1. The processor 2 executes the display task which takes many times as on a real platform.

The Performance Analyzer tool can also evaluate the global average time of the hardware overhead which is about 10.48 cycles. One cycle is the clock period of the HwRTOS. We also refined the overhead per service (Figure 12). The maximum cycle is for the Periodic Delay services (min: 6 cycles, average: 20 cycles, max: 36 cycles).

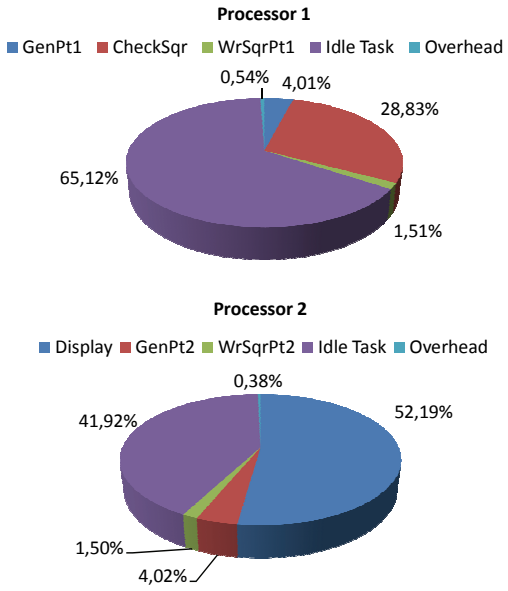


Figure 11 : Distribution of Charge on processors.

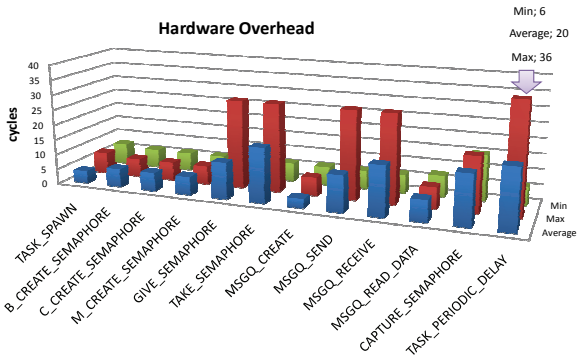


Figure 12 : Hardware Overhead by OS services.

The software overhead corresponds to the interrupt running on a processor.

On the Prototyping board, we have also evaluated the overhead of the application. The hardware overhead does not change because the HwRTOS is a cycle accurate model. On the other hand, the average of the software overhead is about 107 cycles (1.07us). So, the global overhead is about 118 cycles (1.18 us) which is less than classical RTOS.

We have also tested this application on Global Scheduling with migration on the Virtual Platform and Prototyping Platform with success. The cost of the migration (migration overhead) can be added on the Virtual Platform to be close to the real platform. In the example, we copy the binary code of migration task (*CheckSqr* task) on the local memory of processors, at the same address. This drawback is the increase of binary code but the performance is better because it is possible to preempt at any time the task and the cost migration is null because the context of the task is stored on a shared memory (Figure 9).

6.2 Second Example

The second example is a classical application to illustrate a scheduling analysis. We define four independent tasks: T1 and T2 runs on processor 1 and T3 and T4 runs on processor 2. The scheduling algorithm is based on the priority of tasks and has been described in SystemC language instead of VHDL language. We use the configuration of the Virtual Platform with only two virtual processors. As shown the Table 1, we obtain accurate results on the Virtual Platform. We notice the difference between the result of Theory and Virtual Platform is due to the overhead of the HwRTOS. The results have been evaluated with the Performance Analyzer tool. The Performance Analyzer tool is able to load a trace to show precision and accurate behavior of the application thanks to the hardware *Debug* module.

We can also develop global scheduling algorithm and use the built-in task migration of the HwRTOS and then analyze the result thanks to the Performance Analyzer. The application can be executed on the Prototyping platform.

Proc	(Priority, Period, wcet)	Theory	Virtual Platform
Proc 1	Task 1 (20, 100, 25)	25%	25,2%
	Task 2 (30, 90, 50)	55,55%	55,54%
	Idle	19,45%	18,3%
	Overhead	0%	0,96%
Proc 2	Task 3 (25, 100 , 20)	20%	20,19%
	Task 4 (35, 90, 55)	61,11%	61,05%
	Idle	18,89%	17,93%
	Overhead	0%	0,83%

Table 1: Distribution of Charge of the Periodic Tasks Example.

7 Conclusion and Perspectives

This article proposes a multiprocessor Hardware RTOS. The advantages of this HwRTOS are the possibility of co-development of the Multiprocessor SoC and software part with the help of the Open Virtual Platform, the single description of the application with standard primitive (VxWorks like), a single hardware *Debug* module running inside the HwRTOS to capture performances and finally a standard integration of the HwRTOS IP on EDA tools.

Thus, the multiprocessor Hardware RTOS is efficient (fast) and flexible as well since it provides much freedom to the designer for customization. It integrated main OS services with multiprocessor characteristics.

In future works, the objectives will be to add new services to handle hardware tasks on the Virtual Platform and in a SoPC platform through the partial dynamic reconfiguration technique.

References

[1] P. Kuacharoen, M. Shalan and V. Mooney. A Configurable Hardware Scheduler for Real-Time Systems. s.l. : Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'03) pp.96-101, June 2003, 2003. Vol. pp.96-101, June 2003.

[2] K. Baskaran, W. Jigang, and T. .Srikanthan. A Hardware Operating System based Approach for Run-time Reconfigurable Platform of Embedded Devices. s.l. : 6th Real Time Linux Workshop (Singapore), Nov 3-5 2004, 2004.

[3] Klevin, Tommy. Get RealFast RTOS with Xilinx FPGAs. XCELL. 2003, 45.

[4] David Kalinsky, Ph.D., Director of Customer Training. How can I Save 30% of my Embedded Software Development Effort? s.l. : White Paper, Enea Embedded Technology, www.ose.com, Date, pp. 2.