HARDWARE IMPLEMENTATION OF AN EARLIEST DEADLINE FIRST TASK SCHEDULING ALGORITHM

Domen Verber

University of Maribor, Faculty of Electrical Engineering and Computer Sciences, Maribor, Slovenia

Key words: Embedded systems, real-time, task scheduling, EDF, FPGA.

Abstract: Task scheduling in complex hard real-time systems produces interference during the normal operation of the application, prolongs the reaction times, reduces throughput, and makes temporal analysis of the system more difficult. A coprocessor performing functions of the operating system is proposed, in order to cope with this. The paper presents the implementation of an Earliest Deadline First (EDF) task-scheduling algorithm with hardware means (specifically with the FPGA device). This solution to a great deal eliminates the impact of the scheduling during the normal application execution. In addition, because it is implemented in hardware, it outperforms any software implementation. Two solutions are presented with different trade-offs regarding execution time and silicon consumption.

Aparaturna izvedba razvrščanja opravil po strategiji najbližjega skrajnega roka

Kjučne besede: Vgrajeni sistemi, realni čas, razvrščanje opravil, EDF, FPGA.

Izvleček: Razvrščanje opravil v kompleksnih sistemih, ki delujejo v strogem realnem času, vpliva na obnašanje med izvajanjem aplikacije, podaljšuje odzivne čase, zmanjšuje prehodnost in otežuje časovno analizo izvajanja opravil. Da bi se temu izognili, predlagamo ko-procesor, ki bi izvajal funkcije operacijskega sistema. V članku je opisana izvedba algoritma razvrščanja po strategiji EDF (po najbližjem skrajnem roku) z uporabo strojne opreme (natančneje na osnovi programirljivih vezij - FPGA). Ta rešitev v veliki meri eliminira vpliv razvrščanja med normalnim izvajanjem aplikacije. Poleg tega je zaradi diskretne aparaturne izvedbe hitrejše od katere koli programske. V članku sta predstavljeni dve rešitvi, ki ustrezata različnim kompromisom glede hitrosti izvajanja in porabe prostora na vezju.

1. Introduction

Embedded computer systems have become an important part of everyday life, and will have more and more influence in the future. They appear in industrial applications, cars, home appliances, entertainment electronics, etc. They are becoming increasingly ubiquitous, and we hardly notice them anymore. Because of such growth in importance, some other previously unobserved aspects of embedded systems are becoming more significant. An increasing number of embedded computer systems are being used in applications, where improper functionality may cause large financial losses or may even endanger human lives. Such systems need to be dependable and must react to events within strict time restrictions. As a state-of-the-art, proper construction of such systems is, as yet, more art than engineering, and the increasing complexities of such systems makes these things even worse (Ebert *et al*. (2009)).

This paper deals with real-time embedded systems. In such systems, the required functionality must be performed in a timely fashion; i.e. regardless of the situation in the system, certain operations must be completed within the predefined time interval. This can only be achieved with proper management of the tasks. Traditionally, this is performed by the operating system. However, proper management of tasks in an embedded system may be a complex and time-consuming procedure that interferes with the normal operation of the system, especially if it is performed on the same processor as the tasks are executed. Because of this, it would be of benefit if such a load could be reduced or eliminated altogether.

One solution is to employ a co-processor that can perform the functionality of the operating system in parallel to the application's execution. The co-processor only interacts with the application when its sequence of tasks needs to be rearranged. Such a solution also decreases the complexity of system design and the analysis of those temporal circumstances that may occur within the system. It also decreases the minimum reaction times and increases the throughput of the system. Nowadays, multi-core processor solutions are available and are becoming more and more popular within embedded systems. Therefore, any implementation where one of the processors' cores is dedicated solely to operating system operations is feasible.

There are two basic kinds of implementation using such a co-processor. Firstly, an additional standard (micro) processor can be used dedicated solely to performing the tasks of the operating system. However, because of the complexities of algorithms for real-time operating systems, it may perform poorly irrespective of its processing power. Although not discussed in this paper, in addition to task scheduling, the co-processor must also perform other functionalities of the operating systems (e.g. tasks

synchronization, inter-task communication, etc.), which also influences their overall performances. Communication with the main processor must be synchronized with the co-processor's activities, which may additionally delay the reaction. Such solutions are well known and have been implemented for some time (Halang (1986), Stankovic (1991), Cooling (1993), and others).

The second solution is to employ hardware implementation that is dedicated to operating system functionalities. This approach has become feasible through advances in solid-state technology, especially FPGAs. It virtually eliminates any impact of the operating system during the normal application execution, for several reasons. Firstly, the operating system's functionality may be divided into independent parts, which are then implemented, in parallel, as separate processes on a single hardware device. Secondly, synchronization with the main processor can also be implemented asynchronously to other functionalities and, thus, introduce much less overhead. In addition, because it is implemented with hardware means, this approach outperforms any software implementation regarding speed. An "OS-on-a-chip" may be used with less powerful processors allowing cheaper and more power-efficient solutions. Operation of such a device can also be formally verified and certified for use at higher safety-integrity levels than the programmed implementation. Furthermore, such a device may serve additional purposes. For example, it can be used as an intelligent I/O device, it may implement communication layers in distributed control systems, etc. An example of this, in combination with software and hardware parts, and the middleware functionality, was studied and successfully implemented in the IFATIS project (IFATIS (2005)). Further improvements are expected using the new approach presented in the paper.

This paper focuses on the hardware implementation of one of the most important parts of an operating system, task scheduling. Nevertheless, this part has the most influence on any temporal behaviour of the systems. The hardware implementation of the algorithm is carried out using a Field-Programmable Gate Array (FPGA) device. However, for the production phase, ASIC or custom-built chips would be more cost-effective.

The first part of the paper introduces the tasks and task scheduling for hard real-time embedded systems. The second part describes the implementation of such algorithm based on the Earliest Deadline First (EDF) scheduling strategy. Two solutions are presented with different trade-offs regarding execution time and silicon consumption are presented.

2. Tasks and task scheduling

An embedded application usually consists of several computing processes or tasks. Typically, there are much more tasks than processing facilities to execute them, and some tasks are executed sequentially on a specific processing unit. In order to execute tasks effectively, a proper schedule (or arrangement) of tasks needs to be found that conforms to certain restrictions. This process is known as task scheduling. Task scheduling can be performed in advance (a priori), for simple and static applications. For example, in a so-called cyclic executive, a set of tasks is executed periodically. During each cycle, all active tasks are executed sequentially. It is in the hands of the developer to choose a period such that all tasks finish execution prior to the start of the next cycle. However, in dynamic systems where the task load frequently changes, this simple scheme is inadequate. A task may be dormant (inactive), ready for execution (active) or being executed on the processor. An active task may be temporarily suspended during execution due to the unavailability of certain system resources or because of the need for synchronizations with other tasks, etc. In this case, the schedules of the tasks must be planned dynamically.

Traditionally, some kind of priority is used to determine which task must be executed next. An active task with the highest priority is always executed first. If needed, tasks with lower priority may be temporarily suspended to allow the running of more important ones. However, those priority-based scheduling strategies are inadequate for hard real-time systems. In such systems, a started task must be finished prior to a certain predetermined deadline, regardless of the conditions in the system. Therefore, a schedule of tasks must be set-up in such a way that all tasks meet their deadlines. This is done by taking into account their execution times and other time-delaying factors in the system. If they exist, such a schedule is called 'feasible'. Several deadline-driven scheduling methods do exist. The so-called rate-monotonic scheduling strategy can be used for those multitasking systems where all tasks are executed periodically over a-priori known periods. Here the deadlines of tasks are matched with their periods and the tasks are then executed according to them: tasks with shorter periods are executed before those tasks with longer ones. Repetitive periods of activation are known in advance for each task. Therefore, these periods may be used as priorities, and can be employed with priority-based operating systems. In their widely-recognised publication (Liu *et al*. (1973)) Liu and Layland show that rate-monotonic strategy yields a feasible schedule if processor utilization is kept below a certain boundary (for instance, for large number of processors, the processor utilization should be below 70%). Another deadline-driven scheduling policy is Earliest-Deadline-First (EDF). It is more flexible than rate-monotonic strategy and can be used in real-case situations where both periodic and a-periodic tasks are present. EDF has also proven to be the best deadline-driven scheduling strategy for single processor systems regarding feasibility and optimality of system utilisation. Because of this, this scheduling strategy was chosen for our research.

2.1 EDF task scheduling

In EDF, the task with the shortest deadline must be executed first. In order to find it, the scheduler must srutinize the

list of ready tasks and find the task with the shortest deadline. This can be represented by a simple pseudo code:

```
min_task_index = 0
min_deadline = ∞
for i=1 to n do
  if taskinfo[i].deadline < min_deadline then
    min_deadline = taskinfo[i].deadline
   min_task_index = i
  end if
end for
```

The *taskinfo* array holds the information of all ready tasks in the system. Each element consists of a task identification number, worst-case execution time, deadline, etc. This procedure must be executed every time a new task becomes active (i.e. ready for execution), suspended or terminated. Termination or suspension of a task, other than that currently running, does not influence the schedule.

However, this is only part of the work that must be accomplished by the scheduler. Another matter is to prove that the schedule is feasible (i.e. that all active tasks will meet their deadlines). For the EDF, the schedule is feasible if the following equation is fulfilled for each active task:

$$a_k \geq \sum_{i=1}^{k} l_i, k = 1,...,n \quad (1)$$

This equation states that the sum of the remaining execution times l_i of all tasks T_i scheduled to run before, and including task T_k, added to the current time, must be less or equal to the absolute time of deadline a_k of task T_k. In other words, the cumulative workload to be performed prior and during execution of task T_k must be completed before its deadline. The tasks are sorted by their ascending deadlines. Again, the condition determined by the formula is static, and must be re-evaluated only when one or more of the tasks change their states.

This schedulability check can be converted into a form with a double nested loop, illustrated using the next pseudo code:

```
cumulative_finish_time = current_time
for i=1 to n do
  for j=i+1 to n do
    if taskinfo[j].deadline < taskinfo[i].deadline then
      swap(taskinfo[i],taskinfo[j])
    end if
  end for
    cumulative_finish_time = cumulative_finish_time+
taskinfo[i].remaining_exec_time
  if cumulative_finish_time > taskinfo[i].deadline then
    raise deadline_violation_error
  end if
end for
```

The first part of the outer-loop is used to sort the data according to the tasks' deadline. As a side effect, the algorithm also puts the task with the shortest deadline in the first place of the array. Therefore, searching for the task with shortest deadline can be combined with a feasibility check. In the second part of the outer-loop, feasibility is tested by first adding the remaining execution time of the current task to the cumulative execution time, and then comparing it to the task's deadline. Because the remaining execution times of separate tasks are represented as relative time intervals, the current execution time is added to the cumulative execution time at the beginning of the code in order to allow for comparison with the deadlines, which are represented as absolute times. It is presumed that the deadlines of the tasks are converted into an absolute form once they become active.

For simplicity, the test whether a task is active or not, is not shown in the code. Usually not all tasks in the system are active (ready to run) at the same time. Inactive tasks should not be considered in the scheduling. The comparison of tasks' deadlines and the feasibility check should only be done if both tasks are active. This can be achieved with additional checks at the beginnings of both loops.

As a drawback, the EDF algorithm described above requires $N^2/2$ iterations (for the N active tasks) which is more than, for example, with priority based task scheduling; in order to find the task with the highest priority, only N iterations are required.

2.2 Modification of the EDF algorithm for hardware implementation

In theory, the basic EDF scheduling algorithm can be easily translated into any hardware synthesis language (e.g. System C or HDL code). However, we found that ordinary in-memory sorting could be very inefficiently implemented in the hardware. Simple hardware circuits are optimal when operating Boolean and integer quantities, therefore, the entire task's information should be transformed into these two types. Furthermore, only simple operations can be usually implemented in the hardware. Complex operations would require several steps for the executing and consuming more hardware resources. Even some integer arithmetic operations (e.g., comparison and addition) would consume considerable amounts of silicon. Therefore, the goal is to minimize the complexity, and reduce the number of steps of the code. In the pseudo-code above, the deadline comparison in the inner-loop requires two read and two write operations in the table (if swap operation is optimized). The basic sort algorithm should be modified in order to reduce the read/write operations One possible modification is shown in the next pseudo code:

```
cumulative_finish_time = current_time
for i=1 to n-1  do
 moved_data = taskinfo[i]
 min_data = moved_data
 min_index = i
 for j=i+1 to n do
  curr_data = taskinfo[j]
  if curr_data.deadline < min_data.deadline then
    min_data = curr_data
```

```
      min_index = j
    end if
  end for
  taskinfo[min_index] = moved_data
  taskinfo[i] = min_data
   cumulative_finish_time = cumulative_finish_time+ min_
data.remaining_exec_time
  if cumulative_finish_time > min_data.deadline then
     raise deadline_violation_error
  end if
end for
```

This code utilizes modified version of a so-called straight selection sorting. The role of the inner-loop is to find the task with the shortest deadline. When such a task is found, it is placed at the beginning of the table. During each iteration of the outer- loop, one element of the array is being sorted. The feasibility check remains the same as before.

Temporary variables such as *min_data*, *min_index*, etc., can be efficiently implemented in the hardware by means of registers. The whole algorithm is now executed as a sequence of simple assignment and arithmetic operations. In the inner-loop only one read operation remains (with an additional read and two write operations outside the loop). Furthermore, because the information needed for the EDF schedulability test is already in the *min_data* variable, an additional read operation can be avoided. The time and memory complexity of this version of the algorithm remains the same as before. However, the total execution time of the outer-loop's body is reduced significantly, and translation into the HDL language is much more efficient.

2.3 Parallelization of the algorithm

Further optimizations of the EDF scheduling algorithm and feasibility test can be achieved if some of the operations are performed in parallel. Obviously, this approach requires multiple processing resources. It cannot be implemented in software using a single processor. However, it can be easily implemented in hardware.

In our case, the first item to be optimized is the inner-loop. If we can eliminate the inner-loop altogether by executing all repeated steps as a single operation, the EDF algorithm may be executed in just N iterations (i.e., the time complexity is O(n)). However, there is data dependency between the loop iterations, e.g., the content of the min_data variable in one iteration of the loop depends on the values from the previous steps. With the given algorithm, the only available optimization technique we can employ is to execute different operation of the loop in pipeline fashion. The pipeline technique is a well-known approach to speed-up instruction execution within the microprocessors. Whilst one piece of information is being processed during one stage, other data element is being processed within another stage. The number of stages depends on the number of different operations to be executed sequentially. Because the speed of the pipeline depends on the slowest stage, it is preferable that the work is divided equally between different stages. Where there is no data dependency between two or more operations, they can be executed simultaneously. For example, in previous algorithm, the write operation and the feasibility check at the end of the outer-loop can be performed in parallel.

An example of such parallelization for 4 tasks is illustrated on Figure 1.

The inner-loop is divided into four stages: data read, comparison, EDF feasibility check, and data write. During comparison of one element of the table, another element can be read. In addition, the feasibility check can be performed at the same time with the write operation. In this way, we can significantly reduce the execution times. However, the time complexity of the algorithm remains the same.

Certain other kinds of sorting should be used for further improvement of the code. We could also eliminate in-memory sorting altogether. One way to do this is to use an auxiliary table where an ordered list of tasks is maintained. This list is incrementally built from data taken from the original taskinfo table; the tasks' data are taken from one table and placed at the proper place in the other one. The pseudo code for this approach would be:

```
for i=1 to n  do
  curr_data = taskinfo[i]
  pos = i
  for j=1 to n-i do
    if tasklist[j].deadline>curr_data.deadline then
      pos = j
      break
    end if
  end for
  for j=n-i downto pos do
    tasklist[j+1]=tasklist[j]
  end for
  tasklist[pos] = curr_data
end for
```

The tasklist table is the auxiliary table previously mentioned. In it, only data relevant to the EDF algorithm is kept. One

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
READ	R0	R1	R2	R3			R1	R2	R3			R2	R3			R3		
COMPARE		C0	C1	C2	C3			C1	C2	C3			C2	C3			C3	
EDF						E					E				E			E
WRITE						W					W				W			W

Fig. 1: Example of pipeline execution of EDF algorithm for four tasks.

element of the taskinfo table is taken at the beginning of the loop. Then, the proper place for this task is determined in the second table by finding the position of the first element with the later deadline. From this index on, we shift all elements of the list to the end of the structure. In this way, we prepare a place in which new data is put.

At first glance, the complexity of this algorithm is greater than the previous one. Indeed, if we implement this algorithm sequentially, more execution cycles probably would be required than before. On the other hand, in this code, both inner-loops only have simple bodies. The first loop has no data dependency in it and it can be easily parallelized by comparison of the current task's deadline with those deadlines already in the table. This can be done in parallel if a comparator is used with each element of the array. At first sight, the second loop has obvious data dependency between the two loop iterations. Some elements of the array are accessed and modified from two loop iterations. However, this part of the code is only the sequential program representation of a shift operation. In hardware architectures, shift registers and queues are frequently used where pieces of data are moved in a similar way. Therefore, the second loop can be implemented by linking the elements of the *taskinfo* table in serial fashion.

This approach also requires modification of the feasibility check. In comparison to previous scenarios, where data is considered to be in ordered fashion, it is now taken from the original table in random order. For the feasibility check, this would require repetitive summation of the remaining execution time according to (1). In order to avoid this, the attributes of the tasklist table have to be expanded with a new variable, which keeps a sum of the remaining execution times for all task prior to and including the current one. This attribute stands for the role of the cumulative_finish_time variable in previous cases. Every time new task information is put onto the list, its remaining execution time must then be added to all following elements on the list. Notably, these are the same elements as shifted before. In order to obtain the total remaining execution time for the current task, its value is summed with the value of the element before it. Because the deadlines are expressed in absolute form and the execution times are relative intervals, they must be properly converted. This can be done by adding absolute current time to the execution time of the first element on the list. Later on, this value will eventually propagate through all elements on the list.

The parallel version of the EDF algorithm may be summarized as a sequence of four steps, repeated for each active task in the system:

Step 1: Compare the deadlines of all elements on the list with the current one and mark elements with a greater deadline.

Step 2: Shift all marked elements to the end of the list by copying all the data. Update (set) the mark for the last element on the list to be included in the next step.

Step 3: Fill the gap with the current task's information. Add the remaining execution time of the current task to the cumulative execution time for all marked elements. If the first element on the list is updated, set its current cumulative execution time to the values of the current system time.

Step 4: Compare the cumulative execution times with the corresponding deadlines and mark all elements where the deadline would be violated.

An additional step that invalidates all elements on the list is also required. However, this step is done only once at the beginning of the process.

Each step has a fixed execution time, therefore in this way, the time complexity of the EDF algorithm becomes O(n). Further improvements are possible if some of the steps are executed in parallel. If we have the resources for fast comparison during actions 1 and 4, each first and the second pair of steps, can be executed in parallel.

3. Experimental hardware implementation of edf algorithm

Two hardware implementations of EDF scheduling algorithm were tested using a FPGA device. In the first experiment, the optimized version of the algorithm was translated into the HDL language by hand. The pipeline approach was used to speed-up the execution. In the second experiment, the version with the sorted list of tasks was implemented by using a digital schematic design. The basic elements of the design were taken from the existing components library. Xilinx's Spartan2E devices were used during the experiments (Xilinx (2009)). This is an entry-level FPGA device with lower operational frequencies and relatively smaller amounts of configurable logical blocks.

The *taskinfo* table was implemented using dedicated memory blocks (called BRAM), which can be found in all modern FPGA devices. BRAM consists of a small amount (several Kb) of memory elements, which can be configured to have differed bus sizes. Several BRAM block were used in parallel to achieve even wider bus sizes. In this way, it was possible to read or write all components of a single *taskinfo* element at the same time. BRAM blocks were also accessible from the experimental platform. There is no direct equivalent of the *for* loop in hardware. However, it is possible to design counters that generate memory addresses according to the code in the algorithm. The arithmetic operations were performed directly using the HDL code in the first case, and with dedicated comparison components in the second case.

The block diagram, when implementing the first version algorithm, is shown in Figure 2.

Deadline and execution times were kept in two dedicated RAM blocks. The loop-generator generates the values of the loop index variables, which are then converted into memory addresses by the *Read* module. *Compare* and

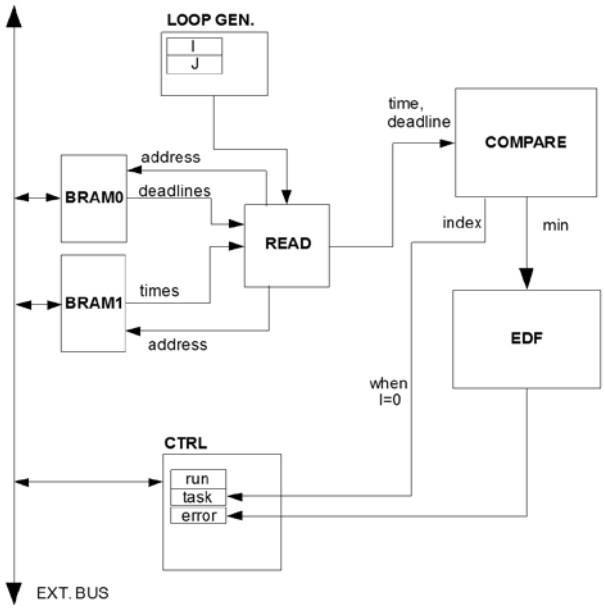

Fig. 2: Block diagram for first implementation of EDF algorithm.

EDF modules perform the comparison and EDF feasibility checks, respectively. The control block is used as an interface between the FPGA and the main processor. Using this block, the main processor starts the EDF scheduling execution. Control block synchronizes the execution of other blocks by properly setting several control signals. The results of the algorithm (index of the task with the shortest deadline and the error status) are also held here.

To allow parallel execution and pipeline implementation, each block is implemented as a set of processes in the HDL code. The HDL version of the loop-generator for loop variable *j* is:

```
– Loop counter j
process (rst,clk,running)
begin
  if rst = ‘1’ or running = ‘0’ then
    j <= “000000”;
  elsif rising_edge(clk) then
    if j = itemcount1 then
      j <= i + “000001”;
    else
      j <= j + “000001”;
    end if;
  end if;
end process;
```

The *itemcount1* variable is set to the number of elements in the table at the beginning. Later, it is decremented within every iteration of the main loop. This variable represents the number of elements that must be processed by the inner loop.

For discrete implementation of the EDF algorithm's second version, for each element in the task list, an appropriate digital logic has been implemented and is represented as a single component. Then several of such components were linked together to implement the task list. This is illustrated in Figure 3.

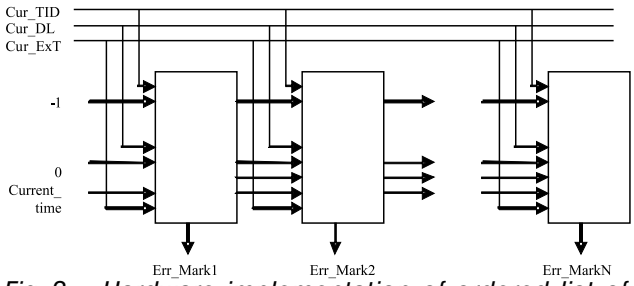

Fig. 3. Hardware implementation of ordered list of tasks' information

In each step, a single task is evaluated and put into the proper position on the list. The information of the current task is put on the common bus. *Cur_TID*, *Cur_DL*, *Cur_Extis* represents the index of the task, its deadline and its remaining execution time, respectively. Then, a series of control signals (not shown in the picture) are generated to execute different steps of the EDF algorithm for each cell, as described in the previous section. The outer-loop generator, the memory blocks and the control logic are not shown. These are similar to the first case.

A logic diagram for each task list element is shown in Figure 4.

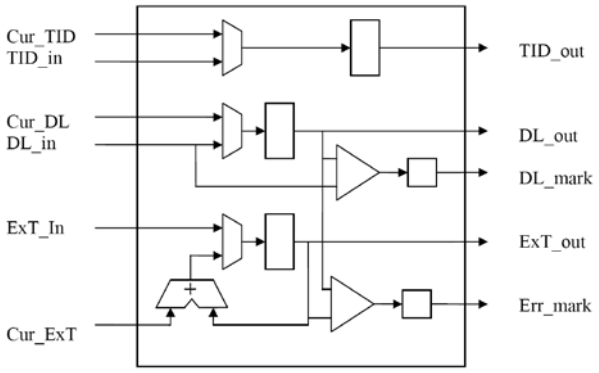

Fig. 4. Logic diagram of the EDF evaluation cell

Each cell has two sets of inputs. One set is fed by the data of the current task and another is feed by data from the previous cell. Two input streams of data are merged into a single output stream to be used during the shift step. The second set of inputs of the first cell is connected to some constant values and to the current time counter, as shown in Figure 3. In this way, no extra logic is required for the first cell on the list. There is one set of outputs connected to the following cell and two logic signals that mark whether the content of the cell should be shifted (*DL_mark*) or if there is a deadline violation error (*Err_mark*).

Each element consists of three registers that contain different attributes of the task list. These registers are connected to the inputs through multiplexers to allow values to be filled-in, either from the common bus or from the previous

cell. Each block is logically divided into three parts. The first (the upper) part tracks the index of the task that is in a particular place on the list. The second (middle) part is responsible for deadline comparison and for marking the cell for the shift operation. The third (lower) part of the cell calculates cumulative execution time and marks deadline violations.

During the first step of the algorithm, the comparator in the middle part of the cell is used to determine the elements with the later deadline (larger value). In the second step, the registers of the marked cells are shifted to the right and the current values are put into the gap. In the third step, the current execution time on the common bus is added to the cumulative remaining execution times of marked cells. This is implemented with dedicated addition components in the third part of the cell. In the last step, the cumulative execution times are compared (with yet another comparator) and any violations are signalled. By using both phases of the clock signal, we managed to reduce the execution time of the steps to two clock cycles of the FPGA device. Finally, the tasks are sorted in ascending fashion according to their deadlines. For the task identification, their IDs are associated with the task deadlines and sorted accordingly. The ID register of the first cell contains a task identification that must be executed next.

4. Results of the experiments

The first version of the algorithm, with pipeline execution optimization described at the beginning of the chapter, takes approximately 140 slices for its implementation. This number is independent of the number of tasks. In the second approach, each EDF evaluation cell requires approximately 30 slices of FPGA device. However, in this case the amount of silicon depends linearly on the number of tasks to be evaluated. For example, in the case of 32 tasks, approximately 950 slices are required – almost 7 times more than in the first case. The control logic and loop generator takes an additional 12 slices. This number only slightly increases if more tasks are evaluated. These results are in favour with the first solution if the amount of silicon resources is limited.

On the other hand, the execution time of the first algorithm for 32 tasks is about 600 basic clock cycles and only around 65 in the second one. The second approach is almost 10 times faster than the first one. Only if the number of tasks is small (less than three), the execution times of both solutions become roughly the same. Therefore, the second approach should be considered if hardware resources are not a problem.

5. Conclusion and future work

Current state-of-the-art technology allows for the hardware implementation of software algorithms, even for low-cost embedded system solutions. Operating systems have well-defined and relatively limited sets of functionalities. Therefore, it is easy to imagine having an "OS-on-a-chip" solution that may be used in the same way as mathematical co-processors two decades ago or as graphical coprocessors are used today. They may even become a part of general processors in the future.

In our future work we will try to achieve the implementation of an EDF algorithm with fixed time execution independent of the number of task (i.e., we want to achieve the *O(1)* temporal complexity). This is possible if the outer loop is eliminated altogether. Thiscan be achieved if the *taskinfo* array is maintained as a sorted list continuously, instead of rebuilding it each time a new task becomes active. However, in this case, in addition to task activation, other task operations performed by the operating system must be considered. First, when a task ends its execution, it must be removed from the array. The tasks following it must be shifted towards the beginning of the array. In addition, if a task becomes suspended due to some synchronization mechanism, it must remain on the table but it must not be considered for the running on the processor. Furthermore, some counters for the remaining execution times of the tasks in the list must be updated periodically. All of this significantly increases the silicon consumption and the feasibility of the approach may be questionable.

References

/1/ Liu C.L. and Layland J.W. (1973). Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM,* 20(1), 46–61.

/2/ Halang, W. A. (1986). Implications on Suitable Multiprocessor Structures and Virtual Storage Management when Applying a Feasible Scheduling Algorithm in Hard Real-Time Environments. Software–Practice and Experience, 16(8), 761-769

/3/ Stankovic, J. A. and K. Ramamritham (1991) The Spring Kernel: A New Paradigm for Real-Time Systems. IEEE Software, 8, 62-72.

/4/ Cooling J. (1993) Task Scheduler for Hard Real-Time Embedded Systems. Proceedings of Int'l Workshop on Systems Engineering for Real-Time Applications. IEE, Cirencester, London. 196-201.

/5/ IFATIS (2005). IFATIS - Intelligent Fault Tolerant Control in Integrated Systems. *http://www.ist-world.org/*

/6/ Ebert C. and Salecker J. (2009). Embedded Software-Technologies and Trends. *IEEE Software*, 26(3), 14-18.

/7/ Xilinx (2009). *http://www.xilinx.com*

Domen Verber

University of Maribor, Faculty of Electrical Engineering and Computer Sciences, Maribor, Slovenia (e-mail: domen.verber@uni-mb.si)

Prispelo: 19.11.2010 *Sprejeto: 24.11.2011*