A Hardware Architecture for Scheduling Complex Real-Time Task Sets*

Sergio Sáez*, Joan Vila*, Alfons Crespo* and Angel Garcia+

*DISCA, Universidad Politécnica de Valencia, Spain
+Departament of Electrical Engineering, Universidad del Valle, Cali, Colombia

The problem of jointly scheduling both hard deadline periodic tasks and soft aperiodic tasks has been the subject of considerable research in real-time systems. One of the most widely accepted solutions for this problem are slack stealing algorithms. However, these algorithms are rather impractical, since they all imply a considerable scheduler overhead. This paper faces the overhead problem by introducing a complete hardware architecture that implements slack stealing in hardware using an optimal algorithm redesigned to be implemented efficiently in hardware. The proposed solution is a circuit that behaves as a kind of sophisticated interrupt controller taking the task workload and the interrupts as inputs, and providing the highest priority task to be executed in the CPU. From the point of view of hardware design, the algorithm involves two main problems: first, to select the highest priority task at every moment and, second, to locate a set of slack gaps in a real-time computation.

Locating slack gaps in a real-time computation is a problem that requires to "look forward in time" into the forecast schedule of a given workload. This paper analyses the different approaches for solving this problem and presents a novel architecture to solve it efficiently using a technique based on an event-driven simulation of the future of a real-time computation. A timing analysis of the proposed design is also presented.

1. Introduction

The workload of a real-time system can be expressed, in general, as a task set composed by a mixture of periodic, and aperiodic tasks. The problem of jointly scheduling hard periodic tasks and soft aperiodic tasks has been subject of considerable research in the last years. As a result, there are several solutions with different performance/cost ratios. Among the proposed solutions [1, 29] are the background server, the polling server, bandwidth preserving servers [12, 27, 28, 8, 6] and slack stealing algorithms [11, 17, 29, 20]. Most of the proposals that claim to be optimal, are highly complex and would imply a significant temporal execution overhead. The main reason for this overhead is the need to look forward into the periodic tasks schedule in order to locate the processor quanta (gaps) where aperiodic tasks can be executed. An approach to reduce this overhead is to provide the run-time scheduler with some tables, elaborated off-line, with the locations of the processor gaps. This method has two disadvantages: first, the spatial cost of the tables and, second, the processor time that is wasted in the (usual) case when tasks have an execution time that is lower than its nominal WCET (worst-case execution time).

The approach followed in this paper to reduce the scheduling overhead is to do scheduling in hardware. The selected algorithm for its hardware implementation is a variation of the Dynamic Slack Stealer by [20] that has been specially adapted to be implemented in hardware [23]. This algorithm is optimal in the sense of minimising the response time of aperiodic tasks without jeopardising the deadlines of periodic tasks. The paper presents a complete hardware implementation of the EDF scheduler and the DSS algorithm. As it is a completely on-line version of the algorithm, no pre-calculated slack table is needed and, therefore, the spatial com-

*This work was supported by the Spanish Government Research Office (CICYT) under grant TIC99–1043–C03–02.

A previous short version of the paper appeared in *Proceedings of the IEEE International Symposium on Industrial Electronics*, Bled, Slovenia, July 12–16, 1999.

plexity of such slack tables is avoided [29, 20]. Furthermore, all dynamic workload variations, such as periodic or sporadic tasks with stochastic execution times (*gain time*), can be taken into account when slack time is calculated.

Most of the literature about hardware implementation of schedulers comes from the field of packet scheduling and packet multiplexing in real-time networks. In these systems scheduling is always done in hardware since efficiency is crucial. Packet scheduling in real-time networks is mostly based on priorities, as it happens in processors scheduling, but the Rate Monotonic (RM) theory is not so straightforward to apply in this case [26]. The key aspect of the hardware implementation of priority policies is the hardware design of priority queues. Static priorities scheduling can be implemented with a fixed number of FIFO queues, one for each priority level. An efficient application of the RM theory requires as many as 256 priority levels. The paper by Moon, *et al.* [15], reviews several architectures for implementing priority queues in hardware and includes a comparison of the four main existing approaches: binary trees of comparators, priority encoders with multiple FIFO lists, shift registers and systolic arrays. An alternative scheme is [9] that uses an associative memory (CAM) to store priority information, and RAM for data storage. Most of these works deal with fixed priorities. The complexity of hardware implementations significantly increases in the case of dynamic priorities, since it requires a scheme for updating priorities and for reordering of packets on a per cycle basis. That can severely degrade the performance of conventional priority queue design. The performance of a given queue design for a particular problem has been analysed throughout different papers.

Some examples can be found in [18] addressing the design of real-time router using a comparator tree, in [13] presenting an implementation called Rotating Priority Queues (RPQ) that provides an efficiency similar to EDF scheduling with a complexity of RMS, in [31] addressing the problem of updating priorities in Fair Queueing algorithms, in [10] presenting a conceptual multi-channel EDF queue for ATM switching and in [16] presenting a novel VLSI Priority Packet Queue (PPQ) that achieves fast operation by manipulating packets instead of isolated words. Some papers also address the problem of full queues [25, 16].

In the field of real-time processing, hardware scheduling is not so usual as in packet switching, but there are also some proposals of real-time coprocessors. The ATAC coprocessor [22] provides support for Ada tasking including scheduling, precise delay implementations, and interrupt handling. Scheduling is based on the RM theory and provides 64 priority levels and priority inheritance for shared objects. Colnaric and Halang [5] introduce the idea of a kernel coprocessor that is responsible for all operating system services. The design is structured into two layers: a Primary Reaction Layer, that handles all external events, and a Secondary Reaction Layer, responsible for operating system requests. From the point of view of scheduling, dynamic scheduling (EDF) is implemented in hardware [4], since it offers a number of advantages over RM, as discussed in [7]. Finally, the Spring kernel [30] introduces a sophisticated coprocessor for a dynamic distributed real-time system where no periodic workload is known in advance. It provides support for multiprocessor scheduling (completely based on heuristics), feasibility checking and task migration [14]. This paper assumes a different model and presents hardware support for a system where the static workload, formed by a set of periodic tasks, is scheduled using EDF and the dynamic workload, modelled as aperiodic tasks, is scheduled using a slack stealer. The main contribution of this scheme is how to perform slack stealing in hardware.

The rest of the paper is organised as follows: section 2 presents the basic hypothesis and problem formulation, section 3 revises the method for slack stealing, section 4 introduces the hardware architecture and the design of its building blocks, section 5 discusses the problem of the clock frequency and, finally, section 6 concludes and points out future work.

2. Problem Formulation

Given a real-time system with a workload defined by a set system of independent periodic tasks $\mathcal{T}$, a stream of sporadic tasks $\mathcal{S}$ (aperiodic tasks with hard deadlines) and a stream of

aperiodic tasks $\mathcal{J}$ with no deadlines, to be executed on a uni-processor system, the goal of the paper is to design a hardware circuit that schedules this workload in the way that minimises the response of aperiodic requests and accepts, whenever possible, sporadic tasks without jeopardising the deadlines of periodic tasks.

The periodic task $\mathcal{T}$ is defined by $\mathcal{T} = \{T_i(C_i, D_i, P_i) : i = 1 \cdots n\}$ with $1 \leq C_i \leq D_i \leq P_i$, where C_i, D_i and P_i are the worst-case execution time, relative deadline and period of the task T_i, respectively. The task set $\mathcal{T}$ is assumed to be feasible [21].

The sporadic task set $\mathcal{S}$ can be defined as $\mathcal{S} = \{S_i(A_i, C_i, D_i) : i = 1 \geq i\}$ with $1 \leq C_i \leq D_i$, where A_i, C_i and D_i are the arrival time, worst-case execution time and deadline of sporadic task S_i, respectively. We assume that the arrival time A_i of each sporadic task is unknown, and that C_i and D_i become known at A_i upon the arrival of S_i. At time A_i, the task S_i must be accepted or rejected if its deadline cannot be guaranteed.

Similarly, the aperiodic task set $\mathcal{J}$ can be defined as $\mathcal{J} = \{J_i(A_i, C_i) : i \geq 1\}$, where the definition and assumptions for J_i are the same that as for S_i, but taking into account that J_i has no deadline, and it cannot be rejected.

The workload at a given instant I_0 is represented by a set of active tasks $\mathcal{A}(I_0)$ that defines the *outstanding computation*, a set of inactive periodic tasks $\mathcal{T}(I_0)$, and the current aperiodic task queue $\mathcal{J}(I_0)$. $\mathcal{A}(I_0)$ is composed of all periodic activations and already accepted sporadic tasks that are unfinished by time I_0.

The tasks do not suspend themselves or synchronise with other tasks and they are ready for execution as soon as its activation occurs.

The circuit that performs the required schedule for the described workload behaves as a kind of sophisticated interrupt controller. The hardware/software interface is defined as follows:

- On startup time, the software writes all the task attributes (C_i, D_i, P_i) into the hardware registers.
- When a sporadic task arrives, the hardware must perform a feasibility test in order to accept or reject the task if its deadline cannot be guaranteed.
- When an aperiodic task finishes its execution or a periodic task suspends itself until the next period, the software writes the task id of that task into the hardware.
- When a context switch should occur, the hardware issues an interrupt and provides the software with an output register that indicates the task id to be executed next.

The controller maintains all the task sets, and calculates the slack gaps and it provides an interrupt only when necessary. The software component of the scheduler is, thus, reduced to the minimal expression, acting only as CPU dispatcher.

3. Algorithm for Dynamic Slack Stealing

This section presents a brief overview of the algorithm for implementing Dynamic Slack Stealing (DSS) algorithm in hardware. The foundations of this algorithm are in the analysis of EDF scheduling by Ripoll *et al.* [21].

The first concept that needs to be introduced is the definition of *slack gap*.

Definition 3.1. For a given feasible task set $\mathcal{T}$, the slack gaps are the intervals of idle time in the schedule of $\mathcal{T}$ that hold when active tasks of $\mathcal{T}$ are processed as late as possible.

Slack gaps were first characterised by Chetto and Chetto [3], for task sets with deadlines equal to periods. Ripoll *et al.* [20] showed the slack time characterisation for periodic tasks with deadlines lower than periods. In their work, a formal method to construct the list of slack gaps is presented. This analysis introduces two functions $G_{\mathcal{T}}(t)$ and $H_{\mathcal{T}}(t)$ which are key to the whole development. These two functions will have to be calculated in hardware in the design presented in this paper.

- **Function** $G_{\mathcal{T}}(t)$: Given a task set $\mathcal{T}$, function $G_{\mathcal{T}}(t)$ accumulates the amount of computing time required by all activations of tasks in $\mathcal{T}$ from time zero until time t. Formally:

$$G_{\mathcal{T}}(t) = \sum_{i=1}^{n} C_i \left\lceil \frac{t}{P_i} \right\rceil$$

- **Function** $H_{\mathcal{T}}(t)$: Given a task set $\mathcal{T}$, function $H_{\mathcal{T}}(t)$ is the amount of computing time required by all activations of tasks in $\mathcal{T}$ whose deadline is less than or equal to t. Formally:

$$H_{\mathcal{T}}(t) = \sum_{i=1}^{n} C_i \left\lfloor \frac{t + P_i - D_i}{P_i} \right\rfloor$$

  In other words, $H_{\mathcal{T}}(t)$ represents the amount of computing time that the scheduler should have served until time t in order to meet all deadlines.

For the sake of clarity, these functions are defined for a synchronous task set $\mathcal{T}$, where all the periodic tasks start at time zero, but they can be easily extended to use $\mathcal{T}(I_0) \cup \mathcal{A}(I_0)$ [19].

Figure 1 shows functions $G_{\mathcal{T}}(t)$ and $H_{\mathcal{T}}(t)$ for the task set $\mathcal{T} = \{T_1 = (1, 3, 6), T_2 = (4, 10, 10), T_3 = (4, 10, 17)\}$. Note that $G_{\mathcal{T}}(t)$ is a stepped function with steps in the beginnings of new periods (0, 6, 10, 12, 17, 18, 20, ...) while $H_{\mathcal{T}}(t)$ is also a stepped function with steps in deadlines (0, 3, 9, 10, 15, 20, 21, 27, ...). Note also that $G_{\mathcal{T}}(t) \geq H_{\mathcal{T}}(t)\ \forall t$.

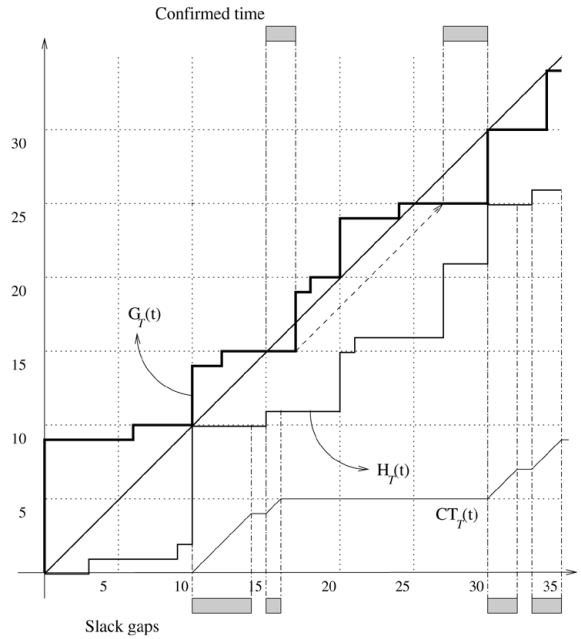


Fig. 1. $G_{\mathcal{T}}(t)$ and $H_{\mathcal{T}}(t)$ Examples.

Using these functions, slack time can be characterised as follows:

Lemma 3.1. For a given feasible task set $\mathcal{T}$, the slack time at time I_0, $ST_{\mathcal{T}}(I_0)$, can be obtained as:

$$ST_{\mathcal{T}}(I_0) = \min_{\forall t \geq I_0} (t - H_{\mathcal{T}}(t))$$

For any feasible set of periodic tasks $\mathcal{T}$ scheduled according to any optimal preemptive scheduler, the slack time $ST_{\mathcal{T}}(I_0)$ represents the maximum time that can be used to service aperiodic tasks until time I_0, without jeopardising the hard deadlines of the periodic tasks.

Once the slack time has been formally characterised, the next step is to obtain a list of slack gaps. An important property shown in [20] is that these gaps always start where $H_{\mathcal{T}}(t)$ changes its value (a step in $H_{\mathcal{T}}(t)$). This is a necessary (but not sufficient) condition for the beginning of a slack gap. Let V_i be the instants where $H_{\mathcal{T}}(t)$ takes a step. To confirm that some V_i is the beginning of a slack gap, it has to be checked that:

$$\nexists V_j : (V_j > Vi) \wedge \Big((V_j - H_{\mathcal{T}}(V_j)) < (V_i - H_{\mathcal{T}}(V_i)) \Big)$$

This requires to search forward into the values of $H_{\mathcal{T}}(t)$ to confirm the beginning of a slack gap. Fortunately this search is bounded, as it will be shown shortly.

Another interesting aspect is the length of a slack gap. Let β_i be the beginning of a slack gap. The length of slack β_i requires to know the beginning of the next gap, that is β_{i+1}, and is given by:

$$\text{length}(\beta_i) = \Delta_{i+1} - \Delta_i \text{ with } \Delta_i = \beta_i - H_{\mathcal{T}}(\beta_i)$$

If the algorithm is looking for an amount S_0 of slack time, its confirmation time is defined as:

Definition 3.2. For a given feasible task set $\mathcal{T}$, and an amount of slack time S_0, the confirmation instant $CI(S_0)$ is defined as: $CI(S_0) = \min(t) : [t > 0] \ \wedge \ [t - G_{\mathcal{T}}(t) = S_0]$

and this confirmation time is bounded, provided this slack time exists.

Lemma 3.2. For any feasible task set $\mathcal{T}$, and an amount of slack time S_0, it can be asserted that: $\nexists t : [t > CI(S_0)] \ \wedge \ [t - H_{\mathcal{T}}(t) < S_0]$

This property indicates that the minimum of the function $t - H_{\mathcal{T}}(t)$ is reached before $t - G_{\mathcal{T}}(t)$ matches the current minimum, and therefore it limits the search range for the confirmation condition.

These definitions show how a list of slack gaps can be constructed for a given task set $\mathcal{T}$. Equivalent definitions can be done, but using the task set at given instant I_0, $\mathcal{T}(I_0) \cup \mathcal{A}(I_0)$.

Consider the above example shown at figure 1 where a periodic task set $\mathcal{T} = \{T_1 = (1, 3, 6), T_2 = (4, 10, 10), T_3 = (4, 10, 17)\}$ is represented. For such task set, the instant of time where $H_{\mathcal{T}}(t)$ takes steps are: $V_i = (0, 3, 9, 10, 15, 20, 21, 27, ...)$ The beginnings of slack gaps are: $\beta_j = (10, 15, 30, 33, ...)$, and the list of slack gaps, with their corresponding lengths: $\theta = \{(10, 4), (15, 1), (30, 2), ...\}$. The grey boxes at the bottom of the figure show where the slack gaps are located (θ).

4. Hardware Design

According to the previous section, it can be stated that the goal of a hardware coprocessor for slack stealing will be to compute the start and the length of the slack gaps in a real-time computation in order to satisfy the requirement for a given amount of slack time. As shown, that requires to search forward in time, looking for the instants of time that meet the slack time characterisation. Once a gap is suspected, the search has to continue until reaching the confirmation time, but fortunately the search depth is bounded. Suspecting and confirming slack gaps require to compute functions $G_{\mathcal{T}}(t)$ and $H_{\mathcal{T}}(t)$ for every instant of time that is inspected.

This section describes the architecture to perform this search into future instants of time of a real-time computation. Two approaches have been devised to achieve this goal:

Tick-oriented. In this approach time is incremented by one in each iteration, checking for the start of gaps and the confirmation of gaps. So, finally, all future instants of time up to confirmation time are inspected.

Event-oriented. This approach is based on the idea that $G_{\mathcal{T}}(t)$ and $H_{\mathcal{T}}(t)$ are stepping functions, so it is not worth checking every future instant of time, but only those where $G_{\mathcal{T}}(t)$ and $H_{\mathcal{T}}(t)$ change value. These changes can be characterised by some well known events: $H_{\mathcal{T}}(t)$ takes a step when the deadline of some task is reached (so it evolves to inactive state) while $G_{\mathcal{T}}(t)$ changes when a task changes to ready state.

In our research both approaches have been studied. The conclusion is that tick-oriented architectures have the advantage of being more simple (they can be implemented using a pipelined binary tree for example [2]) but they also have important drawbacks: the performance and the maximum depth search they can reach analysing future instants of time strongly depends on the granularity of the real-time clock. So if the unit of time is changed, say from milliseconds to microseconds, the implementation will perform 1000 times slower. On the other hand, event-oriented architectures result in a greater complexity, due to event detection, but they are much more powerful. This section concentrates on the design and implementation of a event-oriented architecture for slack stealing in hardware.

The goal of the hardware design is to minimise the processor time wasted by the scheduler and interrupt handling which always results in delays and utilisation reduction.

The scheduler design has been split into a hardware component and a software component, but the last one can be reduced to the minimal expression. Their goals are:

- The hardware maintains all the task sets, calculates slack gaps and informs the software component when a task reaches the maximum priority or an aperiodic task should start execution.
- The software component is only a CPU dispatcher. It only needs to schedule the task indicated by the hardware, and inform the hardware when the current task finishes.

The proposed architecture for the hardware scheduler is shown by figure 2, and its main functions are:

1. To select the highest priority task at every moment. In this case, the scheduling policy follows the EDF basis, and therefore, the highest priority task is the ready task with the earliest deadline.
2. To calculate the $H_{\mathcal{T}}(t)$ and $G_{\mathcal{T}}(t)$ functions, as intermediate step towards aperiodic task scheduling.
3. To calculate the set of slack gaps, in order to know when an aperiodic task can be scheduled or when an sporadic task can be accepted.

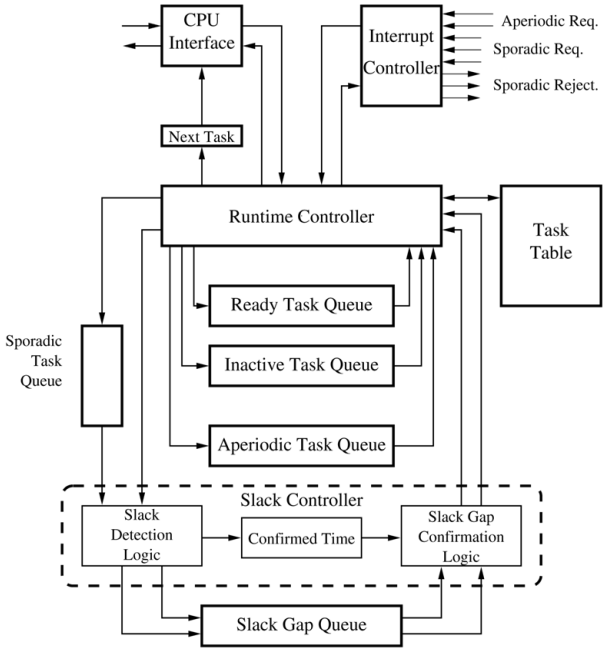


Fig. 2. Hardware Architecture.

Before describing this architecture, it is worth noting that there are two clocks involved in this hardware design:

Real-Time clock (RTC) The execution times are measured using this clock and it determines the processor quantum.

Hardware clock (HC) It determines how fast the hardware scheduler executes its internal algorithms, and it is implementation dependent.

The main components of this architecture are related bellow:

Task table It is a memory that maintains the parameters of the tasks, such periods, deadlines, worst-case execution times (wcet), remaining execution time (ret), etc.

Ready tasks queue (hereafter RTQ) It contains all active tasks ($\mathcal{A}(I_0)$ set) sorted according to the EDF policy. The active task with the highest priority is always at the head of the queue. It is also used to calculate the $H_{\mathcal{T}}(t)$ values.

Inactive tasks queue (hereafter ITQ) It contains the periodic inactive task set ($\mathcal{T}(I_0)$) sorted by activation time. The head of the queue is the next periodic task to be promoted to ready state. It is also used to calculate the $G_{\mathcal{T}}(t)$ values.

Aperiodic tasks queue It contains the aperiodic tasks sorted according to the selected policy (FIFO, *shortest job first*, etc.). If that policy is preemptive, the run-time controller is informed whenever a preemption occurs between aperiodic tasks.
The aperiodic task queue can be implemented using well known static priority queues.

Run-time controller It is the main component of the system. It performs the scheduler role, selecting the highest priority task from the active tasks queue and generating an interrupt towards the CPU whenever a context switch is required. It also calculates functions $H_{\mathcal{T}}(t)$ and $G_{\mathcal{T}}(t)$ necessary for slack detection.

Sporadic tasks queue It is a small queue that sorts all concurrent sporadic arrivals. The sporadic tasks are ordered according to the EDF policy, using a well known static priority queue.

Slack gaps queue (hereafter SQG) It stores the slack gaps (θ), calculated from the V_i and $V_i - H_{\mathcal{T}}(V_i)$ values.

Slack controller It calculates the slack gaps using the slack gaps queue. The required values of the $G_{\mathcal{T}}(t)$ and $H_{\mathcal{T}}(t)$ are proportioned by the run-time controller. It also maintains the amount of slack time that has been confirmed and when it has been confirmed (S_0 and $CI(S_0)$).

Interrupt controller It receives aperiodic and sporadic hardware requests, and informs the RT controller about these requests. If an sporadic request is rejected by the slack controller, it is conveniently signalled..

The main roles are carried out by the dynamic queues (RTQ and ITQ), the run-time controller and the slack controller. Their behaviour and implementation details are described next.

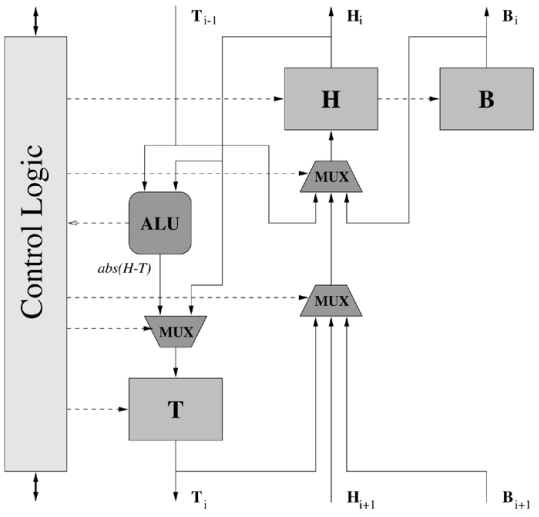
4.1. Dynamic Priority Queues

In this work dynamic priority queues are used for implementing the ready task set $\mathcal{A}(I_0)$ (RTQ), and sorting the future activations of periodic tasks, i.e., the inactive periodic task set $\mathcal{T}(I_0)$ (ITQ). Several hardware structures have been proposed for implementing these queues, such as binary trees of comparators, shift registers or systolic queues. But, if the task priorities are dynamic, i.e., they depend on current time, an implementation problem arises: the task priorities should be updated every clock tick, and the highest priority should be reevaluated. This problem is not so important under the EDF approach, since the head of the list can only change when new periodic activations occur or current running task finishes. According to this, several possible designs are possible:

- To update task priorities (deadlines) only when periodic task activates or the current running task finishes.
- To use absolute values for deadlines, i.e., deadlines values are related to a given fixed instant, called *zero*.
- To use relative values for deadlines, i.e, the value stored in every element is relative to the previous one and only the value at the head of the list is related to the current time.

In the first approach the update overhead could be not negligible if the task set is large enough. The second approach requires wider registers and also introduces the overflow problem at the deadline registers, but it was successfully used at [10]. To avoid such problems, this paper advocates implementing in hardware a priority queue with relative values similar to those queues used by the operating systems to handle multiple clock timers. With this approach, updating deadlines only requires to update the head of the queue.

Implementation details Implementation of the above queue solution has been done using the systolic approach. This solution has the advantage that scales to a large number of periodic tasks and priority levels. Furthermore, it also allows to obtain the highest priority task in constant time.




```
1.  case Ctr_{i-1} of
2.    when Insert =>
3.      T_i.e := abs (H_i.e - T_{i-1}.e);
4.      if (T_{i-1}.e < H_i.e) then
5.        T_i.i := H_i.i;
6.        T_i.f := H_i.f;
7.        H_i := T_{i-1};
8.        Ctr_i := Shift;
9.      else
10.       T_i.i := T_{i-1}.i;
11.       T_i.f := T_{i-1}.f;
12.       Ctr_i := Insert;
13.     end if;
14.   when Shift =>
15.     T_i := H_i;
16.     H_i := T_{i-1};
17.     Ctr_i := Shift;
18. end case;
```

Fig. 3. Dynamic Priority Queue Cell and Algorithm (Insert Command).

A dynamic priority queue is a systolic chain of *event cells*, each of them containing registers H, Ctr, B, and T. Register H is a compound register that stores the following values: a time event value e that represents a deadline for RTQ or a task activation for ITQ, and is relative to the previous cell, a task identifier i or pointer to the task table, a flag f indicating if it is a current event or a future one. Register Ctr is a command control register. Register B is a backup register for storing H value before the start of a simulation and restoring it later (see below). Register T is a temporal register necessary for pipeline systolic behaviour. In addition, each cell contains an ALU to compute $abs(H - T)$, i.e., the difference of $H.e$ with respect the former cell. The algorithm shown in figure 3 describes, in VHDL notation, the insertion behaviour of a event cell. The length of ITQ should be equal to the maximum number of periodic tasks, and the length of RTQ should be equal to the maximum number of periodic and sporadic tasks that can be active simultaneously.

4.2. Run-Time Controller

The main function of the run-time controller is to update the RTQ and ITQ when a CPU quantum ends and to calculate $H_{\mathcal{T}}(t)$ and $G_{\mathcal{T}}(t)$ functions to detect slack gaps. The values of these functions will be provided to the slack controller.

Updating RTQ and ITQ implies that when an inactive task reaches its period, the RT controller promotes it to the ready task set, and when the CPU informs that the current task has finished,

```
1.  if (clock mod 2 = 0)
2.     min_RTQ := (RTQ.H.e <= ITQ.H.e);
3.     if (min_RTQ) then
4.        min_time := RTQ.H.e; min_ident := RTQ.H.i;
5.        if (RTQ.H.f = future)
6.           func_H := func_H + TT[min_ident].wcet;
7.        else
8.           func_H := func_H + TT[min_ident].ret;
9.        end if;
10.       RTQ.Ctr := Extract;
11.       ITQ.H.e := ITQ.H.e - min_time; ITQ.Ctr := None;
12.    else
13.       min_time := ITQ.H.e; min_ident := ITQ.H.i;
14.       execution_time= TT[min_ident].wcet;
15.       ITQ.Ctr := Extract;
16.       RTQ.H.e := RTQ.H.e - min_time; RTQ.Ctr := None;
17.    end if;
18.    future_time := future_time + min_time;
19.    SC.T := future_time; SC.H := func_H; SC.G := func_G;
20. else
21.    if (min_RTQ) then
22.       ITQ.H.e := TT[min_ident].period;
23.       ITQ.H.f := future; ITQ.H.i := min_ident;
24.       ITQ.Ctr := Insert; RTQ.Ctr := None;
25.    else
26.       func_G := func_G + execution_time;
27.       RTQ.H.e := TT[min_ident].deadline;
28.       RTQ.H.f := future; RTQ.H.i := min_ident;
29.       RTQ.Ctr := Insert; ITQ.Ctr := None;
30.    end if;
31. end if;
```

Algorithm 1. $H_{\mathcal{T}}(t)$ and $G_{\mathcal{T}}(t)$ Calculations.

the run-time controller extracts it from the ready tasks queue and, if it is periodic, it inserts a new instance into the inactive tasks queue.

Calculating the values of the $H_{\mathcal{T}}(t)$ and $G_{\mathcal{T}}(t)$ is done by simulating future states of the RTQ and ITQ. In order to maintain the current state of those queues ($\mathcal{A}(I_0)$ and $\mathcal{T}(I_0)$), a backup system should be incorporated to the dynamic queue design.

The $H_{\mathcal{T}}(t)$ and $G_{\mathcal{T}}(t)$ calculation section of the run-time controller is described by algorithm 1 using a notation close to VHDL. The algorithm notation is as follows:

RTQ.H: register H (time event) of the head of RTQ

RTQ.Ctr := Extract: Apply command Extract to the control register of the head of RTQ.

TT$[n]$.wcet: register wcet (worst case execution time) of entry n of the task table.

SC.T: Register T of interface with the slack controller.

min_RTQ: true when the closest event is a task deadline.

The rest of the notation is self explanatory.

This algorithm simulates the future states by updating RTQ and ITQ at the speed of the hardware clock. Obtaining a new state requires two clock cycles. During the first cycle, the heads of RTQ and ITQ are inspected to find out the nearest event (deadline or activation) and the corresponding cell is dequeued. During the second cycle, a cell dequeued from RTQ during the first cycle is inserted in ITQ and a cell dequeued from ITQ is enqueued in RTQ. According to its definition, function $H_{\mathcal{T}}(t)$ is updated in the first cycle every time the head of RTQ is dequeued. The initial value of $H_{\mathcal{T}}(t)$ is 0. Conversely, $G_{\mathcal{T}}(t)$ is updated every time that the head of ITQ is dequeued. However, note that in this case the calculation is done during the first cycle but the update is done in the second cycle since $G_{\mathcal{T}}(t)$ changes its value at the end of a time interval. The initial value of $G_{\mathcal{T}}(t)$ is the sum of the outstanding execution times of all active tasks.

The simulation of future states is event driven (not tick driven) so the time does not increment uniformly: once an event is processed, time is advanced by the value of the processed event.

The run-time controller also informs the slack controller of the sporadic task arrivals in order to confirm them. All sporadic tasks that are confirmed are inserted into the RTQ, to be taken into account in future calculations. The aperiodic and sporadic tasks can be generated from software components by calling an operating system primitive, or from hardware signals through the *interrupt controller*.

4.3. Slack Controller

The proposed hardware scheduler also calculates the set of slack gaps to know when the aperiodic and sporadic task can be scheduled without jeopardizing the hard deadlines of the periodic tasks.

In order to find the slack gaps, the slack controller uses the functions $H_{\mathcal{T}}(t)$ and $G_{\mathcal{T}}(t)$ provided by the run-time controller, and a special hardware queue, Slack Gaps Queue (SGQ), that calculates and stores the future slack gaps in a systolic fashion.

When a sporadic request arrives, the slack controller extracts the required slack gaps from the slack queue to find out if the sporadic task can be accepted. If the appropriated amount of slack time is available before the sporadic deadline, then the task is granted. If so, the sporadic task

```
1.  if (clock mod 2 = 1) then
2.     Δ_H := T − H;
3.     Δ_G := T − G;
4.  else
5.     if (Δ_G > confirmed_time) then
6.        confirmed_time := Δ_G;
7.     end if;
8.     if (T /= LastT) then
9.        SGQ.TS := LastT;
10.       SGQ.TL := LastΔ_H;
11.       SGQ.Ctr := Insert;
12.    end if;
13.    LastT := T;
14.    LastΔ_H := Δ_H;
15. end if;
```

Algorithm 2. Slack Controller Algorithm.

identifier is sent to the RT controller and inserted into the ready tasks queue, to be taken into account in future calculations.

Whenever a slack gap is reached and outstanding aperiodic computation exists, the run-time controller is asked to generate a context switch interrupt.

The algorithm 2 shows the slack controller in VHDL notation. This algorithm basically tries to confirm slack time using the characterization of definition 2 in section 3. It works in two phases. During the first phase, it computes $t - G_{\mathcal{T}}(t)$, that represents an amount of confirmed slack time, for a new step in $H_{\mathcal{T}}(t)$. In the second phase it determines the maximum of this function and inserts into the SGQ a new cell with the values of t and $t - H_{\mathcal{T}}(t)$ for the recently processed step.

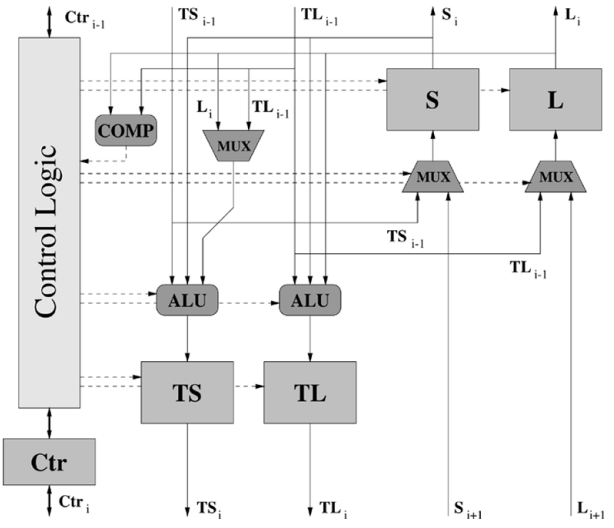
Implementation details of Slack Gaps Queue The SGQ is also implemented as a systolic queue, where each *slack cell* stores a slack gap, i.e. its start time S_i and its length L_i, and a control command register, Ctr. Those values are relative to the end of the slack gap stored in the previous cell, except for the first cell.

The cell and the algorithm of this queue are shown in figure 4. The input values to the SGQ are provided by the slack controller and consist of the current future time V_i, and the function $V_i - H_{\mathcal{T}}(V_i)$ (Δ_i, if V_i is the beginning of a slack gap). These values are inserted in the SGQ as TS and TL respectively. With these input values, the SGQ constructs a list of presumed slack gaps that must be confirmed by using the $G_{\mathcal{T}}(t)$ function.

When the SGQ receives a new slack gap, basically, what every cell does is make it relative to itself and pass a *relative gap* to the next cell. This means $TS_i := TS_{i-1} - (S_i + L_i)$ and $TL_i := TL_{i-1} - L_i$. If, during this iteration, the length of the relative gap becomes zero, then it indicates that the end of the queue has been reached and all outstanding cells are cleared. If, during this iteration, some cell of zero length or a length greater than the relative gap (produced by the previous cell) is found, then this cell sets its length to the length of the relative gap and the queue ends in the next cell.

5. Timing Analysis

Although some other methods, such as complex binary trees, can be used for implementing priority queues and also to calculate $H_{\mathcal{T}}(t)$ and $G_{\mathcal{T}}(t)$ values, the main advantage of the presented method is that it performs an event driven




```
1.  case Ctr_{i-1} of
2.    when Insert =>
3.      if (TL_{i-1} = 0) then
4.        S_i := TS_{i-1};
5.        L_i := 0;
6.        Ctr_i := Clear;
7.      elsif (TL_{i-1} < L_i or L_i = 0) then
8.        L_i := TL_{i-1};
9.        TS_i := TS_{i-1} - (S_i + TL_{i-1});
10.       TL_i := 0;
11.       Ctr_i := Insert;
12.     else
13.       TS_i := TS_{i-1} - (S_i + L_i);
14.       TL_i := TL_{i-1} - L_i;
15.       Ctr_i := Insert;
16.     end if;
17.   when Clear =>
18.     S_i := 0; L_i := 0;
19.     Ctr_i := Clear;
20. end case;
```

Fig. 4. Slack Gaps Queue Cell and Algorithm (Insert Command).

simulation, so it calculates the values of $G_{\mathcal{T}}(t)$ and $H_{\mathcal{T}}(t)$ when they take a step, not continuously. This also avoids the problem with the real time clock granularity. This kind of stepped calculation is difficult to be calculated using other methods, since they would require to compute the step width first, and then the value increment. For example, this calculation would need two steps of $\log_2(N)$ cycles using a binary tree, where N is the number of tasks.

The goal of this analysis is to determine the relation Real-Time Clock / Hardware Clock that allows to locate a given quantity of slack time ST before it is needed. This allows aperiodic and sporadic tasks to be managed as soon as possible. More precisely, in order to obtain the minimum response time for a given aperiodic task A_i, the next slack gap should be located before the start time of the slack gap is reached. To do that, the worst scenario the hardware should be able to face would be to locate a slack time unit in only one real-time clock tick. On the other hand, In order to accept a given sporadic task S_i with a computational requirement of C_i, the quantity of slack time ST found before S_i arrives should be equal or greater than C_i. Otherwise, the task S_i should wait until such quantity of slack time could be found, and then be accepted or rejected.

In order to determine the relation Real-Time Clock / Hardware Clock, it is required to analyze the work to be done in a real-time tick. It can be splitted in two parts:

1. To update RTQ and ITQ according to the Real-Time clock.
2. To extract the slack gaps queue θ by looking forward in time.

According to this, the latency of the hardware algorithm can be represented as:

$$L = t_{update} + t_{extract}$$

The worst case for updating the queues is when all the periodic tasks are inactive and should be promoted from the ITQ to the RTQ. Then t_{update} can be stated as:

$$t_{update} = 2Nt_{HC}$$

where N is the number of periodic tasks, t_{HC} is the period of the hardware clock, and the value 2 comes from the two cycle basis of the run-time controller algorithm.

On the other hand, as it was detailed in [24], the worst case latency for $t_{extract}$ depends on where the worst case confirmation time for a given quantity of slack time ST is located. Such confirmation time $\mathcal{R}$ can be calculated by using the recursive expression $R_{i+1} = G_{\mathcal{T}}(R_i + pS + ST)$ until $R_i = R_{i+1}$, where $R_0 = 0$, and pS is the slack time *used* on the previous RTC tick. The last value of R_i indicates the confirmation time $\mathcal{R}$. For the aperiodic case, pS and ST should be set to 1.

Then, the worst case for extracting ST units of slack time is:

$$t_{extract} = 2At_{HC}$$

where A represents the number of activations and deadlines within the interval $[0, \mathcal{R})$. That is:

$$A = \sum_{T_i \in \mathcal{T}} \left\lfloor \frac{\mathcal{R} + P_i - D_i}{P_i} \right\rfloor + \left\lfloor \frac{\mathcal{R}}{P_i} \right\rfloor$$

6. Conclusions and Future Work

This paper shows the feasibility of implementing a complex scheduling algorithm in hardware, which avoids completely the CPU scheduling overhead. The presented scheduler also shows a good scalability factor due to the systolic design.

The paper describes how to efficiently implement in hardware the following two important problems:

- the systolic priority queues using relative time values, and
- how to anticipate scheduling events using an event-driven approach and avoiding the real-time clock granularity problem.

A timing analysis of the hardware design has also been shown allowing to determine the design suitability for any given task set.

References

[1] N. AUDSLEY, A. BURNS, R. DAVIS, K. TINDELL, AND A. WELLINGS, Fixed priority pre-emptive scheduling: An historical perspective, *The Journal of Real-Time Systems*, 8(2/3):173–198, March/May 1995.

[2] A. G. BAÑOS, *Arquitecturas Hardware para Planificadores de Tiempo Real*, PhD thesis, Universidad Politécnica de Valencia, 1999. in Spanish.

[3] H. CHETTO AND M. CHETTO, Some results of the earliest deadline scheduling algorithm, *IEEE Transitions on Software Engineering*, 15(10):1261–1269, 1989.

[4] M. COLNARIC, D. VERBER, R. GUMZEJ AND W. HALANG, Hardware-supported real-time operating system kernel, *Microprocessor and Microsystems*, 18:579–591, 1994.

[5] M. COLNARIC, D. VERBER, R. GUMZEJ AND W. HALANG, Implementation of hard real-time embedded control systems, *Real-Time Systems Journal*, 14(3):77–94, 1998.

[6] T. GHAZALIE AND T. BAKER, Aperiodic servers in a deadline scheduling environment, *The Journal of Real-Time Systems*, 9:31–67, 1995.

[7] W. HALANG AND A. STOYENKO, *Constructing Real-Time Predictable Systems*, Kluwer Academic Publishers, Boston-Dordrecht-Lond, 1991.

[8] N. HOMAYOUN AND P. RAMANATHAN, Dynamic priority scheduling of periodic and aperiodic tasks in hard real-time systems, *The Journal of Real-Time Systems*, 6:207–232, 1994.

[9] T. HSU AND L. KUNG, A hardware mechanism for priority queue, *Computer Arcitecture News*, 7(66):162–169, December 1989.

[10] B. KIM AND K. SHIN, Scalable hardware earliest-deadline-first scheduler for atm switching networks, in *Proceedings of Real-Time Systems Symposium*, pages 210–218, 1997.

[11] J. LEHOCZKY AND S. RAMOS-THUEL, An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems, in *Proceedings of the Real-Time Systems Symposium*, pages 110–123, 1992.

[12] J. LEHOCZKY, L. SHA, AND J. STROSNIDER, Enhanced aperiodic responsiveness in hard real-time environments, in *Proceedings of the Real-Time Systems Symposium*, pages 261–270, 1987.

[13] J. LIEBEHERR AND D. WREGE, *Design and analysis of a high performance packet multiplexer for multiservice networks with delay guarantee*, Technical report, Department of Computer Science, University of Virginia, 1995.

[14] L. MOLESKY, K. RAMAMRITHAM, C. SHENA, J. STANKOVIC, AND G. ZLOKAPA, Implementing a predictable real-time multiprocessor kernel – the spring kernel, in *IEEE Workshop on Real-Time Operating Systems and Software*, May 1990.

[15] S. MOON, K. SHIN, AND J. REXFORD, Scalable hardware priority queue architectures for high speed packet switches, in *Proceedings of the Real-Time Technology and Applications Symposium*, pages 203–212, 1997.

[16] D. PICKER AND R. FELLMAN, A vlsi priority packet queue with inheritance and overwrite, *IEEE Transactions on VLSI Systems*, 3(2):245–253, June 1995.

[17] S. RAMOS-THUEL AND J. LEHOCZKY, On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems, in *Proceedings of the Real-Time Systems Symposium*, pages 160–171, 1993.

[18] J. REXFORD, J. HALL, AND K. SHIN, A router architecture for real-time communication in multicomputer networks, in *Proceedings International Symposium on Computer Architecture*, pages 237–246, May 1996.

[19] I. RIPOLL, *Planificación Prioridades Dinámicas en Sistemas de Tiempo Real Crĩtico*, PhD thesis, Univ. Politécnica de Valencia, 1996. in Spanish.

[20] I. RIPOLL, A. CRESPO, AND A. GARCÍA-FORNES, An optimal algorithm for scheduling soft aperiodic tasks in dynamic-priority preemptive systems, *IEEE Transactions on Software Engineering*, 23(6):388–400, June 1997.

[21] I. RIPOLL, A. CRESPO, AND A. MOK, Improvements in feasibility testing for real-time tasks, *The Journal of Real-Time Systems*, 11:19–39, 1996.

[22] J. ROOS, Designing a real-time coprocessor for ada tasking, *IEEE Design and Test of Computers*, 8(1):67–79, 1991.

[23] S. SÁEZ, A. GARCÍA, J. VILA, AND A. CRESPO, The real-time stealer, in *Proceedings of the 23rd IFAC/IFIP Real Time Programming Workshop*, pages 61–66, June 1998.

[24] S. SÁEZ, J. VILA, A. CRESPO, AND A. GARCIA, *A hardware architecture for scheduling complex real-time task sets*, Technical Report DISCA-2-98, DISCA, Univ. Politécnica de Valencia, 1998.

[25] L. SHA, R. RAJKUMAR, AND J. LEHOCZKY, Real-time computing with IEEE Futurebus+, *IEEE Micro*, 11:30–33,95–100, June 1991.

[26] L. SHA AND S. SATHAYE, A systematic approach to designing distributed real-time systems, *IEEE Computer*, 26:68–78, 1993.

[27] B. SPRUNT, J. LEHOCZKY, AND L. SHA, Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Proceedings of the Real-Time Systems Symposium*, pages 251–258, 1988.

[28] B. SPRUNT, L. SHA, AND J. LEHOCZKY, Aperiodic task scheduling for hard real-time systems, *The Journal of Real-Time Systems*, 1:27–60, 1989.

[29] M. SPURI AND G. BUTTAZZO, Scheduling aperiodic tasks in dynamic priority systems, *The Journal of Real-Time Systems*, pages 179–210, 1996.

[30] J. STANKOVIC AND K. RAMAMRITHAM, The design of the spring kernel, in *Proceedings of Real-Time Systems Symposium*, 1987.

[31] A. VARMA AND D. STILIADIS, Hardware implementation of fair queuing algorithms for atm networks, *IEEE communications magazine*, 35(12):54–69, 1997.

Received: April, 2000
Revised: June, 2000
Accepted: July, 2000

Contact address:
Sergio Sáez, Joan Vila, Alfons Crespo
DISCA
Universidad Politécnica de Valencia
Camino de Vera 14
46022 Valencia
SPAIN
phone: +34 96 387 95 77
fax: +34 96 387 75 79
e-mail: {ssaez,jvila,alfons}@disca.upv.es

Angel Garcia
Departament of Electrical Engineering
Universidad del Valle
Cali, COLOMBIA
e-mail: angarcia@eiee.univalle.edu.co

SERGIO SÁEZ received the B. S. and Ph. D. degrees in computer science from the Polytechnic Universty of Valencia, Spain, in 1994 and 2000 respectevely. He is assistant professor in the Department of Computer Engineering and Science at the Polytechnic Universty of Valencia. His current research interests include real-time scheduling, multiprocessor systems and hardware-assisted scheduling.

JOAN VILA received the B. S. and Ph. D. degrees in industrial engineering from the Polytechnic Universty of Valencia, Spain, in 1985 and 1994 respectevely. He is professor in the Department of Computer Engineering and Science at the Polytechnic Universty of Valencia. His research interests include distributed systems, real-time communications and real-time systems operating.

ALFONS CRESPO received the B. S. and Ph. D. degrees in electric engineering from the Polytechnic Universty of Valencia, Spain, in 1979 and 1984 respectevely. He is professor in the Department of Computer Engineering and Science at the Polytechnic Universty of Valencia. Since 1988, he has been at the head of lead the Real-Time group, leading several national and European research projects. His areas of technical interests are real-time systems, integration of intelligent components in real-time systems, and real-time operating systems.

ANGEL GARCIA received the B.S. degree in communication engineering in 1985, and the Ph.D. degree from the Polytechnic University of Valencia, Spain, in 1999. He is Professor in the Department of Electrical and Electronic Engineering at the Universidad del Valle, Cali, Colombia. His current research interests include real-time scheduling and hardware/software codesign.