A Binary-Tree Architecture for Scheduling Real-Time Systems with Hard and Soft Tasks

Angel García
Universidad del Valle
A.A. 25360, Cali, COLOMBIA
Phone and Fax: (572) 3392361
angarcia@eiee.univalle.edu.co

Joan Vila, Alfons Crespo, Sergio Sáez
Universidad Politécnica de Valencia
Camino de Vera 14,46022 Valencia, SPAIN
Phone: +34 6 387 95 77, Fax: +34 6 387 75 79
{jvila, alfons, ssaez}@disca.upv.es

ABSTRACT

Complex real-time systems require jointly schedule both periodic task and aperiodic tasks with hard and soft deadlines. This problem has been subject of considerably research in real-time systems and one of the most widely accepted solutions are dynamic slack stealing algorithms (DSS) for scheduling aperiodic tasks, running with earliest deadline first (EDF) algorithms for scheduling periodic ones. However, these algorithms are rather impractical, since they all imply a considerably scheduling overhead that always results in delays and reduced CPU utilization. One of the proposed solutions to this problem is doing scheduling in hardware. This paper follows this approach and analyzes in depth a hardware design based on binary trees. The proposed solution is a circuit that behaves as a sort of sophisticated interrupt controller whose inputs are the task workload and the interrupts, and whose output interrupts the CPU exclusively when a task switch should occur, providing the identifier of the task to be resumed (the highest priority task).

From the point of view of hardware design, two main problems arise: first, to select the highest priority task at every moment using an optimal algorithm like EDF; and second, to locate at each time the nearest slack gap in real-time computation where soft aperiodic tasks can be executed, using a DSS algorithm. Both problems are solved at once, by means of the binary tree architecture proposed.

1. INTRODUCTION

The workload of a real-time system can be expressed, in general, as a task set composed by a mixture of hard periodic and soft aperiodic tasks. The problem of jointly scheduling hard periodic tasks and soft aperiodic tasks has been subject of considerably research in the last years. As a result, several solutions with different performance/cost ratios have been proposed. Among these solutions are the background server, the polling server [1], bandwidth preserving servers [4] and slack stealing algorithms [5, 15, 17]. Most of the proposals that claim to be optimal, are highly complex and imply a significant temporal execution overhead. The main reason for this overhead is the need to look forward into the periodic task schedule in order to locate the processor 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 as disadvantage, the spatial cost of the tables and, second, the processor time that is wasted in the (usual) case that tasks have an execution time that is less than its nominal WCET (worst-case execution time).

Slack stealing algorithms for scheduling aperiodic tasks have been demonstrated to be optimal when used in conjunction with dynamic algorithms for the periodic tasks, like EDF or LLF (least laxity first) [2, 17].

The approach followed in this paper to reduce the scheduling overhead is to do scheduling in hardware. The algorithm selected for aperiodic tasks to be implemented in hardware is a variation of the Dynamic Slack Stealer by Ripoll et. al. [17] that has been specially adapted to perform in hardware [19]. This algorithm is optimal in the sense of minimizing the response time of aperiodic tasks without jeopardizing the deadlines of periodic tasks. As it is a completely on-line version of the algorithm, no pre-calculated slack table is needed, and therefore the spatial complexity of such slack tables is avoided [5, 17]. Furthermore, all dynamic workload variations, such periodic or sporadic

tasks with stochastic execution times (*gain time*), can be taken into account when slack time is calculated. The paper presents a complete hardware implementation of the EDF scheduler and the DSS algorithm.

Several works deal with complex scheduling algorithms implemented in hardware, although most of them are in the field of packet scheduling and multiplexing in real-time networks [8, 11, 14, 16]. Scheduling in these systems is mostly based on priorities, so the key aspect here is the hardware implementation of priority queues. Packet scheduling (as processor scheduling) can be done using static priorities, applying Rate Monotonic (RM), or dynamic priorities (e.g. EDF). RM scheduling policies can be implemented with a fixed number of FIFO queues, one for each priority level. The paper by Moon, *et al.* [13] revises 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 [6] that uses an associative memory (CAM) to store priority information, and RAM for data storage. Most of them are designed for dealing with fixed priorities. However, the complexity of hardware implementations significantly increases in the case 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.

In the field of real-time processing, there have been fewer hardware implementation proposals. A remarkable exception is the Spring kernel [6] where a heuristic-based scheduling coprocessor has been developed to enhance multiprocessor scheduling [12]. However, the assumptions and scheduling solutions adopted are rather different to the ones addressed in this paper.

2. PROBLEM FORMULATION

Given a real-time system with a workload defined by a set system of independent periodic tasks t, and a stream of aperiodic tasks ? with no deadlines, to be executed on a uniprocessor system, the goal of the paper is to design a hardware circuit that schedules this workload in such a way that minimizes the response of aperiodic requests without jeopardizing the deadlines of periodic tasks.

The periodic task t is defined by $t=\{T_i(C_i,D_i,P_i)\}$ with $1 \le C_i \le D_i \le P_i$, where C_i, D_i, P_i are the worst-case execution time, relative deadline and period of the task T_i, respectively. The task set t is assumed feasible [18]. The aperiodic task set ? can be defined as $?=\{T_i(A_i, C_i)\}$, where A_i, D_i are the arrival time and the worst-case execution time of the task T_i, respectively. We assume that A_i is unknown. Tasks do not suspend themselves or synchronize with other tasks and they are ready for execution as soon as an activation occurs.

The workload at a given instant *t* is represented by a set of active tasks that defines the *outstanding computation* and the current aperiodic task queue. Active tasks are all periodic activations unfinished by time *t*.

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

- On startup time, the software writes into the hardware registers all the task attributes (C_i, D_i, P_i).
- When an aperiodic task finishes its execution or a periodic task suspends itself until the next period, the software writes into the hardware the task-id of that task.
- When a context switch should occur, the hardware issues an interrupt that provides the software with an output register that indicates the task-id to be executed next.

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

3. ALGORITHM FOR EDF SCHEDULER

At each time, this algorithm selects to execute the active periodic task with earliest deadline. Deadlines are not constants but they change on time. Although not the unique solution, a binary tree of comparators is appropriate to calculate the minimum deadline of a task set.

4. ALGORITHM FOR DYNAMIC SLACK STEALING

This algorithm searches continuously the future to locate slack gaps that can be assigned to aperiodic tasks. The main operation to do is two additions series, as it be shown shortly. This can be accomplished by means of a binary tree of adders.

First follows an introduction to slack gap calculations. The foundations of this algorithm are in the analysis of EDF scheduling of [18]. The first concept that needs to be introduced is the definition of *slack gap.*

Definition 1 For a given feasible task set t, the background gaps are the CPU idle time, while slack gaps are the intervals of idle time in the schedule of t that hold when active tasks of t are processed as late as possible.

Slack gaps were first characterized by Chetto and Chetto [3], for task sets with deadlines equals to periods. Ripoll *et al*. [17] showed the slack time characterization 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 $G_t(t)$ and $H_t(t)$ 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 $G_t(t)$: Given a task set t, function $G_t(t)$ accumulates the amount of computing time requested by all tasks activations in t from time zero until time *t.* Formally:

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

Function $H_t(t)$: Given a task set t, function $H_t(t)$ is the amount of computing time requested by all activations of tasks in t whose deadline is less than or equal to *t.* Formally:

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

Figure 1 shows functions $G_t(t)$ and $H_t(t)$ for the task set t ={T_1=(2,3,4), T_2=(1,5,8) y T_3=(2,8,12)}. Note that $G_t(t)$ is a stepped function whit steps in the beginning of new periods while $H_t(t)$ is also a stepped function with steps in deadlines. Using these functions, slack time can be characterized as follows:

The amount of available processor time for executing soft aperiodic tasks can be easily deduced from the definition of $G_t(t)$. If $G_t(t) < t$ for some time *t*, then it can be stated that the amount of requested computing time up to *t*, has been less than the available processor time. More precisely, the unused processor time can be expressed as

$$unused(t) = max\left(max_{i\in[0,t]}(i - G_t(i)),0\right)$$

and this is the maximum time than can be used for executing aperiodic tasks in background up to *t*. In the example of Figure 1, $G_t(8)=7$ so *unused*(8)=1. Also, $G_t(12)=10$ so *unused*(12)=2;

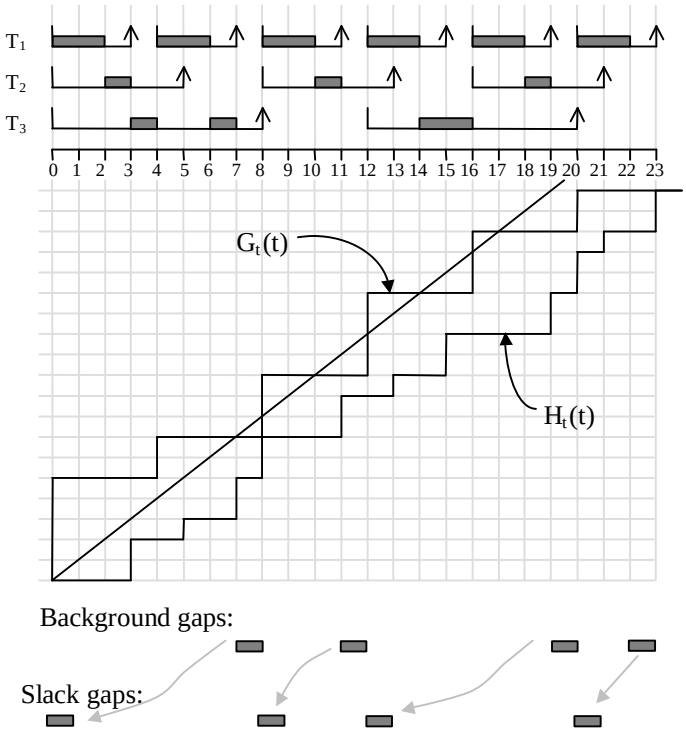

Figure 1: $G_t(t)$ and $H_t(t)$ example

Background gaps are time intervals where aperiodic requests would execute if they would do in background, i.e., while it is not active any periodical task. If *unused*(t-1) < *unused*(t) then the interval [t-1, t] denotes a unitary background gap.

However, it is possible to execute earlier aperiodic tasks. To do this, it is necessary to delay the execution of periodic tasks as much as possible, without violate any deadline. Thus, background gaps shift in time coming near to present. These shifted gaps are named slack gaps.

To determine slack gaps, it has to search backwards in time. Let t_k be a time instant where *unused*(t_k) > 0 and let t_j ($t_j = t_k$) be the earliest deadline when an aperiodic task can be executed making use of *unused*(t_k). Time t_j can be determined searching back in time from t_k for the earliest t_j where the requirement $H_t(t_j)$ + *unused* (t_k) = t_j is met. In the previous example, if t_{k0}=8 then t_{j0}=1. And if t_{k1}=12 then t_{j1}=9.

In other words, function $H_t(t)$ searches candidate slack gaps that must be confirmed by the existence of respective background gaps, discovered by function $G_t(t)$.

Note that both functions $G_t(t)$ and $H_t(t)$ are expressed as addition series of C_i values, which can be calculated appropriately by means of a binary tree of adders.

5. ADAPTING ALGORITHMS TO HARDWARE

An algorithm to attend aperiodic tasks was presented in [17]. The idea was to look forward the background gaps and to advance them in time as long as it is possible without the periodic tasks lose their deadlines. Then a deadline is assigned to each gap, which is placed in the list of tasks to be executed for the EDF scheduler. The advantage of this method is that the new task is integrated from a natural way, without the necessity to make changes in the EDF algorithm. In order to adapt this gap-searching algorithm to hardware, two improvements were included:

- It only looks for the first one-unit gap (which time width is one real-time clock tick), and then a server will assign it to the first aperiodic task that arrives to a queue.
- Instead of carrying out two iterations (the first one looking for background gaps forward in time, with the G function; and the second going back in time from there to calculate the maximum position to which the gap can be moved, with the H function), the whole process is doing in a single forward-time iteration, as stated in [19]. This algorithm is the **gap_search()** function on Figure 2 written in C-seudocode.

```
 long tr, H, G, D_Gap, elapsed;
 bool Gap_accepted, Gap_found;

gap_search()          // CONTROLLER node
{
 initialize();
 while(Gap_accepted == false)
 {
  calculate_H();
  if(tr - H > 0)
  {
   if(Gap_found == false)
    D_Gap = tr - elapsed;
   Gap_found = true;
  }
  else if(tr – H == 0)
   Gap_found = false;
  if(tr – G > 0)
   Gap_accepted = true;
  calculate_G();
  tr = tr + 1;
 }
}

RT_tick_clock()      // CONTROLLER node
{
 if(rising_edge(RT_clock))
  elapsed = elapsed + 1;
}
```

Figure 2: Gap searching

This algorithm calculates the slack gap and its confirmation as background gap, both forward-in-time. At each moment, **D_Gap** variable maintains the candidate deadline of the first one-unit slack-gap found it, while **Gap_found** flag is activate to

validate **D_Gap**. **Gap_accepted** flag is set when the first unit of background gap is reaching. At this moment, the algorithm stops and **D_Gap** holds the minimum deadline that can be assigned to an aperiodic one-unit task without disturbing periodic ones. If background gap is too far away, its searching process may need more than one real-time clock tick. In this case, the calculated deadline must be decremented in **elapsed** units to reflect this time has already elapsed. The **elapsed** variable is incremented every time a real-time clock expires by means of **RT_tick_clock()** function.

```
calculate_H()
{
  for(i = 1; i < NUMBER_OF_TASKS; i = i +1)
  {
    DF[i] = DF[i] - 1;                  // TASK node
    if(DF[i] == 0)                      // TASK node
      if(flag_restart == true)          // TASK node
      {
        flag_restart = false;           // TASK node
        H = H + C_INI[i];               // ALU node
      }
      else
        H = H + CK[i];                  // ALU node
  }
}

calculate_G()
{
  for(i = 1; i < NUMBER_OF_TASKS; i = i +1)
  {
    PF[i] = PF[i] - 1;                  // TASK node
    if(PF[i] == 0)                      // TASK node
    {
      PF[i] = PK[i];                    // TASK node
      DF[i] = DF[i] + PK[i];            // TASK node
      G = G + CK[i];                    // ALU node
    }
  }
}
```

Figure 3: Calculations of G, H, PF and DF

Functions that calculate H and G values are shown in Figure 3. The gap thus found is programmed like any other task, with **D=D_Gap** deadline, with one-unit worst case execution time (**C=1**) and without period (**P=¥**). If, at the moment of its execution, there is not any aperiodic or sporadic pending task, this unitary task will be deactivated and other periodic task will be executed (selected using the EDF policy). Also, the gap searching algorithm will be restarted. And in case there is an aperiodic pending task, the first one in queue will be selected to execution along one unit of time.

Gap searching algorithm takes the supposition that at the hyperperiod beginning the variables are initialized as $H = 0,\ \mathrm{tr} = 0,\ G = \sum_{\forall j} CK_j$. This is correct and easy of doing. However, once the hyperperiod began, with certain frequency will be necessary to restart that algorithm (for instance, if a gap is not timely consumed). In that instant, there will be some tasks in the course of their activation. To restart, **H** function always begin with 0, but **G** function needs to take account the amount of time not yet consumed by each task in the respectively current activation (adding the instantaneous **C** value of each active task). In addition, these instantaneous **C** values (namely **C_INI**) should be stored for each active task, because later will be necessary add them to calculate **H** function, the first time each task finishes its deadline. Inactive tasks at re-starting time do not affect the calculation of the **G** initial value neither the first **H** set up.

The great inconvenience of the restarting is that requires an additional register (to store the **C_INI** value) for each task. This register may be suppressed if anyone of the following conditions is meeting:

- If the algorithm that explores the future looking for gaps restarts every time a pulse of the real-time clock expires. However, this condition limits the search depth.
- If all task deadlines are smaller than the depth that the algorithm can explore to look for gaps in the future, during one real-time clock tick.

In both cases, when the algorithm should use the initial **C_INI** to set up **H**, its value is still identical to **C**. Therefore, we can use this last register, saving us the first one. However, in this implementation the register **C_INI** is not suppressed, in order to not limiting the search depth (Figure 4).

```
initialize()
{
  G = 0;                    // CONTROLLER node
  H = 0;                    // CONTROLLER node
  tr = 0;                   // CONTROLLER node
  elapsed = 0;              // CONTROLLER node
  Gap_found = false;    // CONTROLLER node
  Gap_accepted = false;  // CONTROLLER node

  for(i = 1; i < NUMBER_OF_TASKS; i = i +1)
  {
    DF[i] = D[i];                  // TASK node
    PF[i] = P[i];                  // TASK node
    if(E[i] == READY || E[i] == RUNNING)
    {
      C_INI[i] = C[i];             // TASK node
      flag_restart = true;         // TASK node
    }
    else                           // TASK node
    {
      C_INI[i] = 0;                // TASK node
      flag_restart = false;        // TASK node
    }
    G = G + C_INI[i];              // ALU node
  }
}
```

Figure 4: Algorithm for gap searching restart

6. BINARY TREE ARCHITECTURE

Previous algorithms can be mapped efficiently to a hardware architecture consisting on a binary tree of elementary processors that can execute minimum and add operations. All the processors of the terminal nodes (called **TASK**) are identical and each one maintains up-to-date each data task. Also all the processors of the internal nodes (called **ALU**) are identical and they carry out the calculations in that all tasks are involved. For example, if it is required to schedule four tasks, the binary tree will be as Figure 5 indicates.

There is also a **CONTROLLER** node connected to the computer interface (where the scheduler software side resides). The **CONTROLLER** generates commands to all **TASKs** for its own initiative or as a computer request. Each **TASK** executes those commands and forwards them toward the respective **ALU**. Each **ALU** receives the same commands but different data from the two nodes of the previous level: one from the left and another from the right. In addition, each **ALU** sends commands and data to the node of the superior level. The **ALU** root offers the last result to the **CONTROLLER**. Which parts of the algorithms are executed in which node are explained in previous figures as comments.

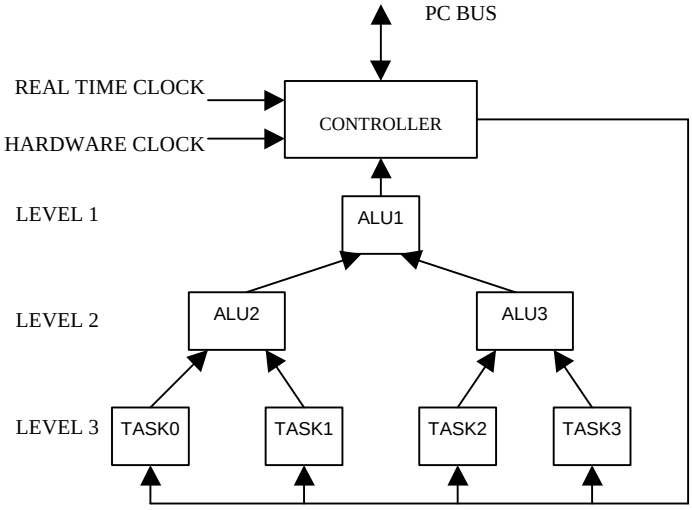

Figure 5: Binary tree for 4 tasks

The communication of commands and data are made by means of a 4-bit serial bus (2-bit for data and 2-bit for control). It was preferred to use a serial bus to facilitate routing connections, since the architecture in binary tree would waste too much silicon area if using parallel buses. In the present architecture, 2-bit data width was chosen (instead of 1) because the commands usually take two data parameters that, in this way, can be processed in parallel, increasing the speed.

In general for **J** tasks, **J TASK** nodes and **J-1 ALU** nodes on $(\lceil LOG_2(J) \rceil + 1)$ levels are required. Each level works as a stage of a pipeline. The **CONTROLLER** distributes this pipeline clock to all **ALU** and **TASK**s nodes by means of an inverse binary tree. This hardware clock determines the speed the architecture executes its algorithms and it is implementation dependent. To avoid electronic-load problems, the connection from the **CONTROLLER** to all **TASK** nodes is also made, in practice, by means of an inverse binary tree, with registers in each **ALU** node, to separate

pipeline stages (this is not indicated in the figure, for not complicating it too much).

Each **TASK** node has three constant 16-bit registers: **PK**, **DK** and **CK** to store period, deadline and the worst-case execution time (w.c.e.t.), respectively, of a task. It also has one status 3-bit register **E**, which holds the state of each task. Moreover, it contains three 16-bit decrementing counters **P**, **D**, **C**, to take account of the period, deadline and w.c.e.t. in each time instant. There is also a 16-bit register **C_INI** to memorize the current value of **C** to restart gap-searching algorithm. Also, there are two 16-bit registers **PF** and **DF** that take account of the period and deadline when searching for a gap in the future. Finally, it contains an 8-bit wired number (**ID**) that identifies each **TASK**.

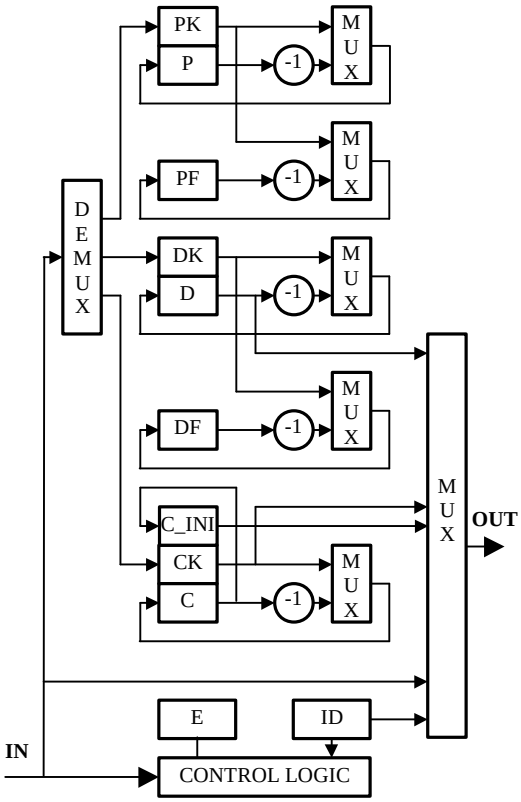

Figure 6: TASK's node blocks diagram

The block diagram of a TASK node is showed on Figure 6. The control logic block decodes the received command and generates the microsignals that control the operation of the other blocks. A decrementer exists for each one of the registers **P**, **D**, **C**, **PF**, **DF**. All arithmetic is 1-bit serial. **TASK** output is selected dynamically by means of a multiplexer (between input command and **ID**, **C_INI**, **CK**, **D** data). This way, it allows to send the command towards the following node and, at the same time, required data is added to it.

The blocks diagram of an **ALU** node is shown on Figure 7. Basically, it carries out two operations, according to the received command, with the data received from the left and from the right:

- Calculation of the smallest deadline and its associated identifier, that receives from the left and from the right.
- Calculation of the sum of partial values H (and G) that receives from left the and from the right.

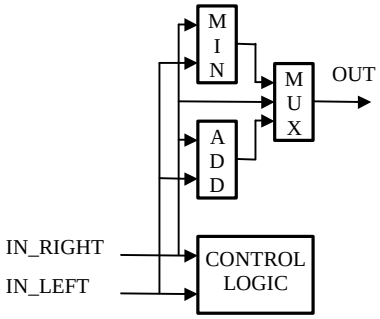

Figure 7: ALU's node blocks diagram

The **CONTROLLER** will receive the final result (minimum deadline and identifier, or G and H values) after crossing $2\lceil LOG_2(J)\rceil$ pipeline stages, being **J** the number of tasks (the 2 factor is due to the inverse binary tree that distributes the commands from the **CONTROLLER** to all **TASK** nodes).

The **CONTROLLER** is the heart of the whole system (Figure 8). Basically it contains five 16-bit registers to interface with the PC (**STATUS**, **PK**, **DK**, **CK**, **COMMAND**); three registers (**tr**, **G** and **H)** which width depends on the number of tasks (i.e. to avoid overflow, 24-bit width are necessary to manage 256 tasks); a little register (**ELAPSED)** typically 3-bits width (it depends on the depth it needs to explore the future for searching gaps); one 16-bit register (**D_Gap**) that stores the one-unit gap deadline when found it; and two flags (**Gap_found** and **Gap_accepted**). Also contains four adders (including two incrementers for **tr** and

elapsed, not shown on the diagram), three substracters as well as several state machines. All arithmetic is serial with 1-bit width.

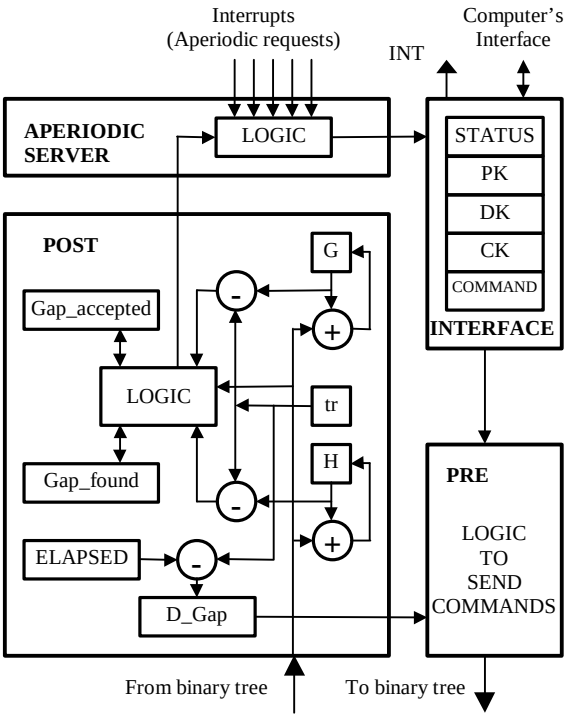

Figure 8: CONTROLLER's block diagram

The **CONTROLLER** should carry out four independent threads (weakly synchronized) implemented in respective subsystems. They are:

- **INTERFACE**: it manages the interface with the computer**.** It receives commands from the computer with their corresponding data (to create, stop and delete tasks) and it allows the computer to read the status register. It sends interruptions to the computer to signal task activations.
- **PRE:** it sends commands to the binary tree to activate the new high-priority task according to EDF; to carry out the accounting of each task; and to ask which is the new high-priority task. It also re-sends the received commands from the computer to the TASK nodes. When suitable, it sends a command for restarting the gap searching. The rest of the time it sends commands continuously to calculate the successive future values of **G** and **H**.
- **POST:** it receives and processes results from the binary tree. The main task is to accumulate the values of **G** and **H** received, to look for the first one-unit slack gap. When found it, a command is issue to create a special task (**TASK 0**) to manage this unitary gap. Register **tr** maintains the "instant of future time" that it is explored.
- **APERIODIC SERVER:** each hardware interruption is associated to an aperiodic task with a known worst case execution time. When EDF policy selects **TASK 0**, the first pending aperiodic task is activated during one unit of real-time clock and its identifier is sent to the computer.

7. OPERATION SPEED

Most of the time, the **CONTROLLER** is sending continuously a command to calculate **H** and **G** values (unless in short moments when other commands are sent). The length of this command depends of the number of tasks. It has 2 header bits, $\left\lceil \frac{LOG_2(J)}{2} \right\rceil$ identifier bits[1] and $16+\lceil LOG_2(J+1) \rceil$ bits for **G** and for **H**.

The main goal of the DSS is explore the future as much as possible searching for a gap. This binary tree architecture needs a time to explore each future pulse of:

$$TC = \left(18 + \left\lceil \frac{LOG_2(J)}{2} \right\rceil + \lceil LOG_2(J+1) \rceil \right) * TX$$

being TX the minimum pipeline period achievable with a certain hardware technology (we are managing times near to TX=30ns, using a XC6216-2 FPGA implementation, but with full custom VLSI design, it will be less). In addition, there are $2\lceil LOG_2(J) \rceil$ levels of pipeline that need $2\lceil LOG_2(J) \rceil * TX$ time to fill it.

Figure 9 shows the speed of this architecture compared for different number of tasks (**J**), in **TX** units. In addition, the last column shows the same delay for a previous hardware architecture build for comparison purposes [20], that implements directly

[1] Half bits are sent by each bus data line (there are 2 bus data lines). G was sent by one data line while H was sent by the other.

the sequential software algorithm, without using parallelism.

J	TC_{BINARY_TREE} (TX ns)		TC
	When loading pipeline	After pipeline is loaded	
2	2	21	24
4	4	22	48
8	6	24	96
16	8	25	192
32	10	27	384
64	12	28	768
128	14	30	1536
256	16	31	3072
512	18	33	6144
1024	20	34	12288
2048	22	36	24576

Figure 9: Time required for exploring one unit of future time

For J=2048, this is three orders of magnitude better than sequential-hardware algorithms (that in turn is much better than software). Better as J increases.

8. MAPPING IN HARDWARE

Binary trees are not well suited to map in two-dimensional silicon area. In general, a simulated annealing algorithm can be employed to reduce area and interconnections length. However when leaf nodes (**TASK**) are much larger than internal nodes (**ALU**), as this case, an optimal regular solution exists.

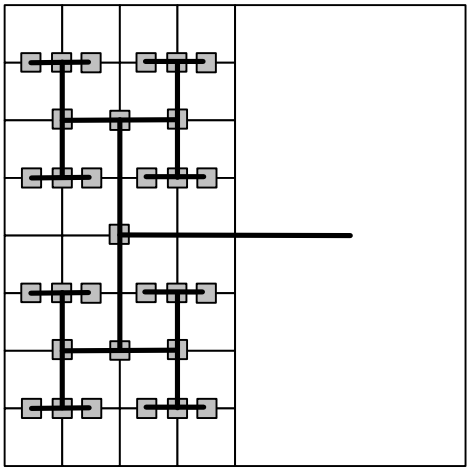

Figure 10: Optimal mapping for binary tree architecture

As shown in Figure 10, the **CONTROLLER** is the largest rectangle to the right, TASK nodes are the medium size squares and ALU nodes are the shadowed small-size ones. Binary tree connections are signaled with a thick line.

9. CONCLUSIONS

The architecture presented based on a binary tree of processors has the following characteristics:

- The time consumed exploring the future searching for a gap is predictable, proportional to $O(\log_2 J)$, as it has just showed.
- It can restart the search algorithm in any moment, with only a low cost associated to fill the pipeline.
- It shows good scalability properties in silicon area and speed:
  - The silicon area grows almost-linear with J. This is because each task needs a **TASK** node and an **ALU** node. Each node does not increase in area due to serial design (except that it is necessary to add 1 bit to the ID register of all TASK nodes, every time the number of tasks duplicates).
  - The speed decreases logarithmically as number of tasks increases, due to two factors: long routing connections between **ALU** nodes and longer commands (to carry longer IDs and serial data).

All this makes feasible real-time schedulers for mixed periodic and aperiodic tasks.

This architecture was designed and simulated on VHDL at three levels (behavior, data flow and structural). There have been carried out enough simulations with several periodic and aperiodic workloads, which shown a good behavior as predicted. Now we are implementing this architecture on an XC6216-2 FPGA-based PC board, for little task sets (we manage an estimated of 8 tasks). With an additional effort, a full custom VLSI design allows us more tasks and more speed.

10. REFERENCES

[1] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, A. J. Wellings, “Fixed Priority Pre-emptive Scheduling: An Historical Perspective”,

Real-Time Systems, Vol.8, No.2/3 March/May 1995, pp.173-198, Kluwer.
[2] T-S. Tia, J. W.-S. Liu, M. Shankar, "Algorithms and Optimality of Scheduling Soft Aperiodic Requests in Fixed-Priority Preemptive Systems", Real-Time Systems, Vol.10, No.1, January, pp. 23-43, 1996, Kluwer.
[3] H. Cheto, M. Cheto, "Some Results of the Earliest Deadline Scheduling Algorithm", IEEE Transactions on Software Engineering, Vol.15, No. 10, pp.1261-1269, 1989.
[4] T. Ghazalie, T. Baker, "Aperiodic Servers in a Deadline Scheduling Environment", The Journal of Real-Time Systems, Vol. 9, pp. 31-67, 1995.
[5] M. Spuri, G. Buttazzo, "Scheduling Aperiodic Tasks in Dynamic Priority Systems", Real-Time Systems, Vol.10, No.2, March 1996, pp.179-210, Kluwer.
[6] D. Niehaus, K. Ramamritham, J. A. Stankovic, G. Wallace, C. Weems, W. Burleson, J. Ko, "The Spring Scheduling Co-Procesor: Design, Use and Performance", Proceedings Real-Time Systems Symposium, pp. 106-111, North Carolina, December 1993.
[7] K. Jeffay, D. L. Stone, "Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems", Proceedings Real-Time Systems Symposium, pp. 212-221, North Carolina, December 1993.
[8] B. Kim, K. Shin, "Scalable Hardware Earliest Deadline First Scheduler for ATM Switching Networks", Proceedings of Real Time Systems Symposium, pp. 210-218, 1997.
[9] S. Sáez, A. García, J. Vila, A. Crespo, "A Hardware Design for a Real-Time Stealer", Technical Report DISCA-1-98, DISCA, Univ. Politécnica de Valencia, 1998.
[10] S. Sáez , J. Vila, A. Crespo, A. García, "A Hardware Architecture for Scheduling Complex Real-Time Task Sets", Technical Report DISCA-2-98, DISCA, Univ. Politécnica de Valencia, 1998.
[11] J. Liebeherr, 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.
[12] L. Molesky, K. Ramamrithnam, C. Shena, J. Stankovic, G. Zlokapa, "Implementing a Predictable Real-Time Multiprocessor Kernel: The Spring Kernel", IEEE Workshop on Real-Time Operating Systems and Software, May 1990.
[13] S.-W. Moon, K. G. Shin J. Rexford. "Scalable Hardware Priority Queue Architectures for High-Speed Packet Switches", Proceedings of the Real-Time Technology and Applications Symposium, pp. 203-212, 1997.
[14] D. Picker, R. Fellman, "A VLSI Priority Packet Queue with Inheritance and Overwrite", IEEE Transactions on VLSI Systems, Vol. 3, No. 2, pp. 245-253, June 1995.
[15] S. Ramos-Thuel, J. P. Lehoczky, "On-Line Scheduling of Hard Deadline Aperiodic Tasks in Fixed-Priority Systems", Proceedings Real-Time Systems Symposium, pp. 160-171, North Carolina, December 1993.
[16] J. Rexford, J. Hall, K. G. Shin, "A Router Architecture for Real-Time Communications in Multicomputer Networks", Proceedings International Symposium, pp.160-171, 1993.
[17] I. Ripoll, A. Crespo, A. García-Fornes, "An Optimal Algorithm for Scheduling Soft Aperiodic Tasks in Dynamic-Priority Preemptive Systems", IEEE Transactions on Software Engineering, Vol.23. No.6, pp. 388-400, June 1997.
[18] I. Ripoll, A. Crespo, A. K. Mok, "Improvement in Feasibility Testing for Real-Time Tasks", Real-Time Systems, Vol.11, pp.19-39, 1996, Kluwer.
[19] S. Sáez, A. García, J. Vila, A. Crespo, "The Real Time Stealer", Proceedings of the 23rd IFAC/IFIP Real-Time Programming Workshop, pp. 61-66, China, June 1998.
[20] A. García, A. González, J. Vila, "Planificador de Prioridades Fijas para Sistemas de Tiempo Real, Implementado en Hardware", VIII Congreso Latinoamericano de Control Automático, Viña del Mar, Chile, Noviembre, 1998.