15th Triennial World Congress, Barcelona, Spain

TASK DECOMPOSITION IMPLEMENTATION IN RT-LINUX

J. Vidal, A. Crespo, P. Balbastre

Departamento de Informática de Sistemas y Computadores
Universidad Politécnica de Valencia
{jvidal, alfons, patricia}@disca.upv.es

Abstract: Control applications require defining several parallel activities to model the environment. Periodic tasks model the activities to be executed at periodic instants of time. While the process of control design is focused on obtaining the regulator, later on translated into an algorithm, the software design is focused on producing pieces of software that will be executed concurrently under a scheduler. Nowadays, more and more applications require complex computation and the use of complex algorithms that can compromise the response time of the system. The activities involving a control loop task can be structured in some parts: data acquisition, computation of the control action, optional activities and output of the control action. This decomposition is useful to improve the control performance and reduce delays due to the scheduler. This paper shows how to implement complex real-time control applications by means of periodic tasks in RT-Linux, using a task decomposition. *Copyright © 2002 IFAC*

Keywords: Real-time system, scheduling, real-time operating systems

1 INTRODUCTION[?]

During the last years an emerging field of integrated control and scheduling has been object of several works. In this field a closer interaction between control design phase and control implementation (scheduling) is used to improve the control performances. The development of scheduling techniques and control theory considering both aspects permits the definition of new flexible scheduling schemes where the control design methodology takes the availability of computing resources into account during the design phase and allows the optimization of control performance and computing resource utilization.

In digital control, it is well known that the system is behaving in open-loop in between two sampling periods. Thus, the control performance degrades as far as the sampling period increases and the degrading depends on the control effort applied to the plant (Albertos *et al.*, 1999). It is also generally the case that delays between the measurement sampling and the control signal updating deteriorates the control performances (Albertos *et al.*, 2000).
A classical limitation in the selection of the sampling period is determined by the complexity of the control algorithm and the time needed by the CPU to compute the result. But, in the case of complex systems, there are many other limitations as the multitasking effects and the computation time variations and the control performances induced by the delays.

To reduce these effects some previous work in the integration of control and scheduling can be found in the literature considering several aspects:

- To increase the use of the CPU adjusting the control

[?] This work has been supported by the Spanish Government Research Office CICYT under grant TIC1999-1226-C02

loops frequency off-line taking into account the process dynamic and the system schedulability (Seto *et al.*, 1998). This method considers the period range of the tasks involved in a control design as criteria to obtain an optimal use of the system resources. In this integrated approach, the implementation is tuned translating a control performance index into task sampling periods to execute the control tasks at the maximum frequency while ensuring the system schedulability. The sampling periods were considered as variables and the method determine their values so that the overall performance was optimized subject to the schedulability constraints. An on-line application of the approach is suggested in (Shin and Meissner, 1999) .

- To reduce the basic timing constraints of a control loop. The timing constraints associated to a control task are the period and the control delays due to the input/output latency. If the delay is fixed and known, the control algorithm can be designed to counteract its effect. From a control point of view, sampling jitter and input-output jitter can be interpreted as disturbances acting on the control system. In (Crespo *et al.*, 1999, Albertos *et al.*, 2000, Balbastre *et al.*, 2000,) a task partitioning scheme is defined to reduce the variable task delays of the control loop implementation.

- To reduce the control performance degrading identifying a parameter as the control effort and adjusting the priority scheme and splitting the task set. A methodology to consider all these aspects has been proposed in (Albertos *et al.*, 2000).

- To dynamically adjust the task periods to maintain the CPU load in bounds. In (Buttazzo *et al.* , 1998) a model based on an elastic task for periodic tasks is presented. Each task has associated an elasticity coefficient and may change its period within certain limits depending on the system load. This approach can be used under fixed or dynamic priority scheduling.

- To consider the on-line use of the CPU as an input to a controller. A feedback controller adjusts the sampling frequencies to maintain the CPU utilization at a desired value. In (Stankovic *et al.* ,1999) it is proposed to use a PID controller as an on-line scheduler under the notion of Feedback Control-EDF. The scheme can be used to adjust the periods or to handle mode changes.

- To handle abrupt variations of the execution time an on-line scheduling feedback control is proposed in (Cervin and Ecker, 2000). The proposed scheme attempts to keep the CPU utilization in a prescribed level avoiding overload situations.

Thus, an important work has been done to integrate both phases, control design and implementation, but it is always assumed that the control algorithm is fixed and only some parameters (period and delay) can be adapted. While the process of control design is focused on obtaining the regulator, later on translated into an algorithm, the software design is focused on producing pieces of software that will be executed concurrently under the supervision of a scheduler. The software designer has to ensure that all the tasks meet their deadlines, i.e., the system is schedulable. Nowadays, more and more applications require complex computation and the use of complex algorithms that can compromise the response time of the system.

In this work, we describe the implementation in RT-Linux of the partitioning scheme proposed in (Crespo *et al.*, 1999) in a efficient way. In Section 2 the periodic task scheme is presented. In Section 3 details the task organization and the basic mechanism used to task synchronization and communication.. Section 4 details the main aspects of the implementation. Section 6 ends with some conclusions.

2. PERIODIC TASK SCHEME

The design of a control system involves the definition of several control loops (each one is a task) that have to be executed under the operating system scheduler. Therefore, periodic tasks are the main components in the design of real-time control applications, performing the actions at regular intervals of time. The periodic scheme is a well known model and there are several methods and techniques for the design, analysis and validation of this systems (Burns and Wellings, 1996).

In (Crespo *et al*., 1999) an partitioning task scheme was proposed. This scheme an improve the control performances reducing the variable delay of all tasks. Under this scheme, the system priorities are split into three bands: final, initial and main bands. The final band covers the highest priorities, the initial band considers the intermediate priorities and the main band the lowest. A periodic job can be split into three tasks: initial, main and final assigning a priority to each part in a the corresponding priority band.

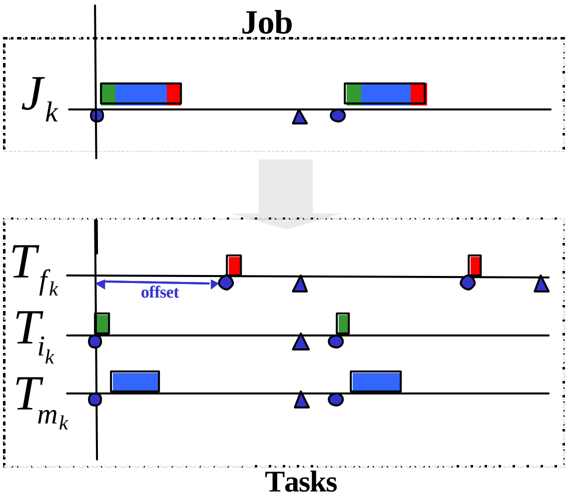

Fig. 1. The initial control activity, Job, is split in three activities.

In (Albertos *et al.*, 2000) is stated the advantages of the variable delay reduction in combination with a control parameter as the control effort.

A job can be decomposed into several tasks taking into account its control activities:

- Mandatory task (M)
- Mandatory and Final tasks (MF)
- Initial and Mandatory tasks (IM)
- Initial, Mandatory and Final tasks (IMF)

However, this scheme presents some drawbacks as the number of task in the system (three times the number of jobs for IMF decomposition). To reduce this drawback we can consider that all final tasks will use the same output devices so, its concurrent execution will be serialized by a resource management. The same reasoning can be applied to initial tasks. In this way, a reasonable solution is to define two server tasks to serve the final and initial activities. These servers will have the following characteristics:

1. Serve all the tasks of a band
2. Have a multiperiod resulting of all the tasks periods in a band
3. Execute the activities of a singular task in each activation
4. Apply the first come first server scheduling policy

From the task synchronization point of view, a final task has to wait its main task ending to be executed. On the other hand, initial and main tasks are executed in the correct order due to their priority assignation.

Figure 2 shows the task queues in the reduced scheme.

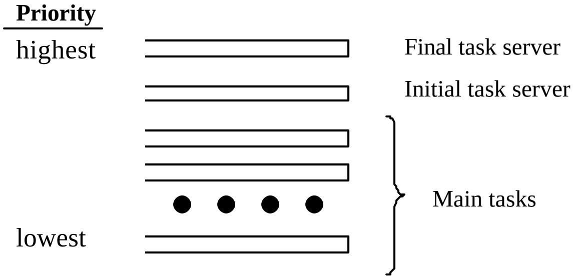

Fig. 2. Priority assignation to tasks

The shedulability analysis of the reduced scheme can be considered the same developed in (balbastre,) considering the union of all tasks served by the respective served using shared resources.

3. IMPLEMENTATION IN RT-LINUX

The conventional approach for allowing real-time features in Unix systems is to modify the kernel in order to make it predictable and to provide an additional set of system calls to gain access to real-time features. This set of system calls has been standardized by POSIX. Unlike this approach, RT-Linux does not modify the Linux kernel or provides additional system calls. Instead of trying to make the kernel of Linux predictable, what it does is to build a kind small microkernel or software layer, called RT-Linux, directly on the bare hardware. Linux runs on top of this layer.

RT-Linux implements the concept of RT-task and uses its own scheduler for these tasks. The default scheduler that comes with RT-Linux is a preemptive, fixed priority scheduler, but it can be easily customized. Linux, including all its kernel and user processes, is regarded by this scheduler just as another RT-task and shares the processor with other RT-tasks (Figure 3). More precisely, Linux is the lowest priority RT-task and thus, it is executed as a background task, so it only runs when no other RT-tasks are running. RT-Linux also provides full control over interrupts, and it implements a software interrupt manager. It allows either: to capture and handle interrupts, or just bypass them to Linux.

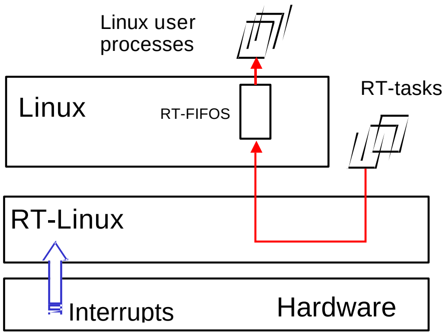

Fig. 3. RT_Linux architecture

The implementation in RT-Linux will consider the creation of two threads for the final and initial server at the highest priority (FINAL_PRIORITY) and the next (INITIAL_PRIORITY) respectively. Each main task will have the priority inside the band. The priority order of main tasks and jobs is kept in the system.

The data structure to store the information of final and initials tasks is the following;

```
// Parameters for IMF tasks
struct server_params{
        int job_id;                // job identifier
        int priority;           // job priority
        hrt_time period, deadline, offset;
        hrtime_t next_activation;
        proc activity;         / code to be executed
};
```

where next_activation will maintain the time for the next task activation and activity is a pointer to the procedure to be executed.

The initial and final servers will use the following variables with the information of all instantiated initial and final tasks

```
server_params  final_server_info [MAX_TASKS];
server_params  initial_server_info [MAX_TASKS];
```

To split a job into its initial, main and final task it is necessary to provide the information associated to the job: job_identifier, period, deadline and offset, the final task offset and the procedures where the initial, main and final activities are coded. The following code shows the implementation filling the data structure of initial and final tasks and creating the thread associated to the main task.

```
int create_IMF_tasks (int job_id,int prio,
     hrtime_t  period, hrtime_t  deadline, hrtime_t offset,
     hrtime_t final_offset,
    proc p_mandatory, proc p_initial, proc p_final)
{
pthread_attr_t attr;
struct sched_param p;
// fill the parameters structure of initial and final .
   add_initial:task(job_id,prior,period,deadline, offset,
                       p_initial);
   add_final:task(job_id,prio,period,deadline,
                     offset+final_offset,   p_final);
  init_semaphore(job_id);
// Creates the thread and make it  periodic
   create_periodic_task(job_id,prio,period,deadline, offset,
                              p_mandatory);
   NTASKS++;
   return 0;
}
```

To split job into **M**, **IM**, and **MF** tasks, the interface is similar including only parameters associated to the decomposed tasks. For example a **MF** decomposition will have the following interface:

```
int create_MF_tasks (int job_id,int prio,
     hrtime_t  period, hrtime_t  deadline, hrtime_t offset,
     hrtime_t final_offset,  proc p_mandatory, proc p_final)
```

The create_periodic_task operation consist in the thread initialisation and the use of the RT_Linux function to do it periodic.

```
// create_periodic_task MACRO CODE.
#define create_periodic_task(job_id,priority,period, deadline,
                              offset, p_mandatory  )
do {
  pthread_attr_t attr;
  struct sched_param p;
  pthread_attr_init(&attr);
  sched_params.sched_priority=priority;
  pthread_attr_setschedparam (&attr, &sched_params);
  pthread_create(&thread,&attr, (void *)
  mand_thread_code,(void *) mandatory,(int *)task_id);
  pthread_make_periodic_np(thread,(RTIME)
                   (initial_time + .offset), (RTIME)  period);
 } while (0)
```

Finally, the mandatory task has the following code:

```
// mandatory task thread code
void * mand_thread_code(void * arg){
long end=ITERS;
int index=(unsigned)arg;
  while(1){
     //execute mandatory under exclusion of final task
    execute_and_signal(sem[index],par[index].activity);
    pthread_wait_np();
    if (end<0) break; end--;
 }// end while
  pthread_exit(0);
  return 0;
}
```

4. SERVER IMPLEMENTATION

A multiperiod server is implemented as a thread that consult the data structure associated to the server to determine the next activation of any of the tasks associated to their service. Each task is considered as a piece of software executed by the server thread. The next activation of the multiperiod server is calculated using the function:

```
void get_next_final_delay(int *id, hrtime_t *delay);
```

which looks for the shortest time to wake up any of the task. Tasks in table are sorted by priority and the operation cost depends on the number of tasks.

Initial and Final servers have different behaviour. While the server of initial tasks has a precedence relation between the initial task and its main task which is solved by design (the priority of the initial is higher than the main, so it always will be executed first), the final server has to handle its precedence relation by means a protocol. The main task has to be finished before the final task can run. The initial server has the following code:

```
void * initial_server(void * arg){
  long end=ITERS;
  int index=(unsigned)arg;
  int id=0; hrtime_t delay;
  int i=0;
  while(1){
    get_next_final_delay(&id,&delay);
    clock_nanosleep(CLOCK_RTL_SCHED,
        TIMER_ABSTIME, hrt2ts(delay), NULL);
    do_initial_action(id); //
  }// end while.
  pthread_exit(0);
  return 0;
}
```

The server of final tasks has to implement a protocol ensuring correct execution order of each pair main and final tasks. This protocol has to consider that when a final task reach its activation time, the main task has to be finished. A semaphore provides the basic mechanism to control this relation. However, if the final task is blocked in the semaphore, it can cause the missing of next activation of other final tasks. So, the access to the semaphore has to be done using a non blocking operation and tests the operation result to see if the operation fails. In this case, the final task is labelled as pendent and executed when the semaphore has been signalled.

```
//final server task code.
void * final_server(void * arg){
  long end=ITERS;
  int index=(unsigned)arg;
  hrtime_t delay;
  int id=0,id_pendent, error;

  while(1){
```

```
   // look for next  final activation.
   get_next_final_delay(&id,&delay);
   // is the next final activation pendent ?
   if pendent(id) {
       // next final activation task is pendent, so
      // looks for the highest priority pendent task.
       id_pendent=get_hprio_pendent();
      // wait for semaphore. Max wait time=delay.
      error=sem_timedwait(sem[id_pendent],
                                   hrt2ts(delay));
     // elapsed time.
     if (error !=0) {
             error=sem_trywait(sem[id]);
             if (error!=0) {
                mark_pendent(id);
             }
             else {
                do_action(id);
                mark_not_pendent(id);
             }
   }
  // next final activation task isn't pendent.
  // so, it is handled consequently.
  else
  {
     // suspend currend until next activation.
    clock_nanosleep(CLOCK_RTL_SCHED,
         TIMER_ABSTIME, hrt2ts(delay), NULL);
    //Wait_for mandatory task activation end
    // wait for semaphore. Max wait time=delay.
    error=sem_timedwait(sem[id],
                                 hrt2ts(delay));
   // elapsed time.
    if (error!=0) {
        mark_pendent(id);
    }
    else
    {
       do_action(id);
        mark_not_pendent(id);
     }

  } // end pendent if.
   //Exit when the simulation ends.
   if (end<0) break; end--;
  }// end while.
  pthread_exit(0);
  return 0;
}
```

The thread associated to the server are instantiated with the next declaration:

```
pthread_t final_server, initial_server;
int Init_final_server(int FINAL_ID, int FINAL_PRIO);
int Init_Iniital_server(int INITIAL_ID,
                             int  INITIAL_PRIO);
```

5. EXAMPLE

In this section, we describe an example to show the partitioning scheme, the split task, the main code, the RT_Linux measures and a snapshot of the real execution.

The system has the jobs described in Table 1.

Table 1 Jobs in the initial control system

Job	WCET	Period	Deadline	Offset
1	6	50	50	0
2	13	80	80	0
3	15	110	110	0
4	16	120	120	0
5	20	200	200	0

The following table (Table 2) shows the use of external interface of each job. A job that does not use to the sensor or actuator means that use internal data as input or output or the jitter does not affect to the control performances.

Table 2 Input/output and job requirements

Job	Sensor	Actuator	Task type
1	Yes	Yes	IMF
2	No	Yes	MF
3	Yes	Yes	IMF
4	No	No	M
5	Yes	Yes	IMF

Using the method proposed in (Balbastre *et al.*, 2000) we obtain the next tasks associated to each job.

Table 3 Job decomposition into tasks

Job	Task	WCET	P	D	O
1	Initial	2	50	50	0
1	Main	2	50	50	0
1	Final	2	50	33	17
2	Main	10	80	80	0
2	Final	3	80	56	24
3	Initial	2	110	110	0
3	Main	10	110	110	0
3	Final	3	110	79	31
4	Main	16	120	120	0
5	Initial	2	200	200	0
5	Main	15	200	200	0
5	Final	3	200	132	68

The set of jobs presents large variable delay (evaluated in terms of DAI and CAI) in input and output showed in third and fourth columns of table 4. After the task decomposition, these variable delays are reduced to the values showed in the last two columns.

Table 4 Variable delays of the job set and the decomposed tasks

	Initial Jobs		Split tasks	
Job	DAI	CAI	DAI	CAI
1	0%	0%	2%	4%
2	-	7.5%	-	8%
3	17.5%	59.6%	12%	4.5%
4	-	-	-	-
5	26.5%	28%	8.5%	5.5%

The next code shows the main module to define and create the task system. The procedures associated to the main, initial and final operations are provided by the control designer.

```
int init_module(void)
{
initialize_threads; // initialize both servers and a thread for each main task.
  int i=0;
   // creates tasks of all jobs
  create_IMF_tasks (1, 10, 50, 50, 0, 12,
                              main1, initial1, final1);
  create_MF_tasks (2, 9, 80, 80, 0, 20,
                              main2,  final2);
   create_IMF_tasks (3, 8, 110, 110, 0, 58,
                              main3, initial3, final3);
  create_M_tasks (4, 7, 120, 120, 0,
                              main4);
    create_IMF_tasks (5, 6, 200, 200, 0, 69,
                              main5, initial5, final5);

    Init_final_server(1, 12);
    Init_Initial_server(2, 11);
     return 0;
}
```

The RT_Linux module obtained by this process design generates a code overhead of 30Kb with respect to the control application. Additionally, the RT_Linux kernel requires 100Kb. From the point of view of number of tasks, this design adds 2 tasks to the initial control software design.

CONCLUSIONS

This paper describes how to implement real-time applications using the decomposition method presented in (Crespo *et al*., 1999). Each of control design activities, called jobs, are divided in mandatory and optional parts, and this allows to reduce the jitter variation (Balbastre *et al.*, 2000). A modular and generic software design has been proposed to implement the decomposed method using RT_Linux. In this implementation every control task (mandatory), is implemented as a thread. Final and initial tasks are grouped and served by two dedicated servers. The implementation of the server has been detailed in the paper. Finally, an example showing the process design and results has been reported.

REFERENCES

Albertos P., Crespo A., Ripoll I., Vallés M., Balbastre P., (2000) "RT control Scheduling to reduce control performance degrading". 39th IEEE Conference on Decision and Control. Australia, December 12-15.

Audsley N., Burns A., Tindell K., Richardson M., Wellings A. (1993) "Applying new schedulability theory to static priority pre-emptive scheduling". *Software Engineering Journal*, 8(5):284-292

Balbastre P., Ripoll I., Crespo A., (2000) "Control Task Delay Reduction under Static and Dynamic Scheduling Policies" 7th International Conference on Real-Time Computer Systems and Applications (RTCSA'00). Cheju Island, South Korea, 12-14 December, 2000

Balbastre, P. And Ripoll I. (2001) Integrated Dynamic Priority Scheduler for RTLinux", Real time Workshop 2001.

Burns A., and A. Wellings (1996), Real-Time Systems and their Programming Languages (2nd Edn), Addison-Wesley.

Cervin A., Ecker. J. (2000)*"Feedback Scheduling of Control Tasks"* Proceedings of the 39th IEEE Conference on Decision and Control, Sydney, Australia, December 2000

Crespo, A., Ripoll I., and P. Albertos. (1999) "Reducing Delays in RT control: the Control Action Interval" *IFAC World Congress,* Beijing[7]Leung J., Whitehead J., "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks", Performance Evaluation, Vol. 2(4), pp. 237-250, December 1982.

Crespo A., P. Balbastre, and S. Terrasa. (2001) "Complex Task Implementation in Ada" Proceedings 6th International Conference on Reliable Software Technologies - Ada Europe 2001, Leuven, Belgium, May 14-18, 2001, Dirk Craeynest, Alfred Strohmeier (Eds.), Lecture Notes in Computer Science, vol 2043, Springer-Verlag, 2001

Liu C.L. and J.W.Layland. (1973) "Scheduling algorithms for multiprogramming in a hard real-time environment". *JACM*, 20,46-61.

Seto D., J.P. Lehoczky, L. Sha. (1998) "Task Period Selection and Schedulability in Real-Time Systems". *IEEE Real-Time Systems Symposium*

Shin K. and C. Meissner (1999) "Adaptation of control system performance by task reallocation and period modification" In Proceedings of the 11th Euromicro Conference on Real-Time systems, pp. 29-36