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.

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...
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 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:
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. 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.

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:

```c
// Parameters for IMF tasks
struct server_params{
    int job_id; // job identifier
    int priority; // job priority
    hrt_time period, deadline, offset;
    hrttime_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

```c
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.

```c
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, prio, 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:

```c
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.

```c
// 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:

```c
// mandatory task thread code
void * mand_thread_code(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.

```c
//final server task code.
void * final_server(void * arg){
    long end=ITERS;
    int index=(unsigned)arg;
    hrtime_t delay;
    int id=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.
// 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 current 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_initial_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

<table>
<thead>
<tr>
<th>Job</th>
<th>WCET</th>
<th>Period</th>
<th>Deadline</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>50</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>80</td>
<td>80</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>110</td>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>120</td>
<td>120</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>200</td>
<td>200</td>
<td>0</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Job</th>
<th>Sensor</th>
<th>Actuator</th>
<th>Task type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Yes</td>
<td>Yes</td>
<td>IMF</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>Yes</td>
<td>MF</td>
</tr>
<tr>
<td>3</td>
<td>Yes</td>
<td>Yes</td>
<td>IMF</td>
</tr>
<tr>
<td>4</td>
<td>No</td>
<td>No</td>
<td>M</td>
</tr>
<tr>
<td>5</td>
<td>Yes</td>
<td>Yes</td>
<td>IMF</td>
</tr>
</tbody>
</table>

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

Table 3 Job decomposition into tasks

<table>
<thead>
<tr>
<th>Job</th>
<th>Task</th>
<th>WCET</th>
<th>P</th>
<th>D</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Initial</td>
<td>2</td>
<td>50</td>
<td>50</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Main</td>
<td>10</td>
<td>100</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Final</td>
<td>3</td>
<td>50</td>
<td>56</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>Final</td>
<td>10</td>
<td>80</td>
<td>80</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Main</td>
<td>10</td>
<td>110</td>
<td>110</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Final</td>
<td>3</td>
<td>110</td>
<td>79</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>Initial</td>
<td>16</td>
<td>120</td>
<td>120</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Main</td>
<td>15</td>
<td>200</td>
<td>200</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Final</td>
<td>3</td>
<td>200</td>
<td>132</td>
<td>68</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Job</th>
<th>Initial Jobs</th>
<th>Split tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DAI</td>
<td>CAI</td>
</tr>
<tr>
<td>1</td>
<td>0%</td>
<td>0%</td>
</tr>
<tr>
<td>2</td>
<td>7.5%</td>
<td>8%</td>
</tr>
<tr>
<td>3</td>
<td>17.5%</td>
<td>59.6%</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>26.5%</td>
<td>28%</td>
</tr>
</tbody>
</table>

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


