THE BUDGET MANAGEMENT FOR FAILURE PREVENTION IN REAL-TIME SYSTEMS

Lamine BOUGUEROUA\(^{(1,3)}\),
Laurent GEORGE \(^{(3,4)}\), Serge MIDONNET \(^{(1,2)}\)

\(^{(1)}\)ESIGETEL, 1 port de Valvin, 77210 Avon
\(^{(2)}\)IGM, Université de Marne la Vallée, Cité Descartes, 5 Bd Descartes, Champs-sur-marne
\(^{(3)}\)Univ. Paris 12, LIIA, 120 rue P. Armangot, 94110 Vitry sur Seine
\(^{(4)}\)ECE, LACSC, 53, rue de Grenelle, 75007, Paris

\{Lamine.Bougueroua, Serge.Midonnet\}@esigetel.fr, lgeorge@ece.fr

ABSTRACT

This paper presents a mechanism for execution-overrun management in real-time systems. It is defined as the time budget a faulty task can use in case of execution overrun without compromising the real-time constraints of all the tasks in the system. This mechanism enables to cope with execution overrun before a deadline miss. Budget use the allowance notion which depends on free resources. We show how to determine the budget value and how to use this mechanism into fixed priority scheduled real-time system (FP/HPF).

KEY WORDS: fault management, allowance, budget management, real-time scheduling, feasibility analysis, fault tolerance, FP/HPF.

1. Introduction

The correct dimensioning of a real-time system depends on the determination of Worst Case Execution Duration (WCET) of the tasks. Based on WCET, a feasibility condition \([1]\) can be established to ensure that the deadlines of all the tasks are always met, whatever their release time. Computation of the WCET can be performed either by analyzing the code on a given architecture or by measurement of the execution duration \([2]\). In both cases, the correctness of the WCET is hard to guarantee in the presence of hardware or software faults. Furthermore, the obtained WCET can be pessimistic compared to the average execution duration, leading to a pessimistic dimensioning of the system. The increasing importance of Java in real-time systems also creates new dimensioning problems. Indeed, if we consider a real-time Java environment, the WCET depends on the condition of execution (type of architecture, type of memory or cache) leading to an imprecise WCET, which is possible in a Java system environment where the underlying architecture is not always known in advance. Since faults are unavoidable \([3]\), fault tolerance is necessary.

Generally speaking, a fault-tolerant system can be made up of redundant components so that the system delivers correct services (i.e. is safe) despite faults \([4]\). Faults can be classified according to their duration \([5]\):

- Permanent or intermittent faults.

We consider two types of faults captured in the execution-overrun error and deadline-miss error. In the first case, the execution duration of the task exceeds its WCET while in the second case the task does not meet its deadline.

In the paper, we first reiterate classical concepts and notations in section 3. We then focus on the allowance principle and show how to compute the static allowance \([6]\) in section 4 when tasks are scheduled with fixed priority \([7]\)[8][9]. We show how to optimize the use of free resources by the introduction of the budget concept in section 5. We give simulation results in section 6 showing the benefits of the budget principle. Finally, we conclude.

2. Related work & Objectives

Most fault tolerant real-time systems present solutions to deal with deadline miss by stopping the execution of the tasks that miss their deadlines. In case of overloaded systems, the idea is to stop some tasks to prevent the others from missing their deadlines and to come back to a normal load condition. In case of overload, tasks are scheduled according to their importance to prevent deadline miss cascading effects: Locke \([10]\), Robust EDF \([11]\) and D-Over\([12]\). The problem with this solution is that a task missing its deadline might already have
cascading effects on the other tasks of the system. The reaction might be too late.

In this paper, we propose a fault prevention mechanism to prevent when possible an execution overrun error from leading to a deadline misses error. We introduce the concept of allowance to determine the maximum duration a task can proceed with its execution without compromising the real-time constraints of all the tasks. This solution is based on the observation that a task execution overrun does not necessarily lead to a deadline miss. Indeed, a system with enough free resources (slack time) can self-stabilize without compromising the timeliness constraints of the tasks. Slack time analysis has been extensively investigated in real time server systems in which aperiodic (or sporadic) tasks are jointly scheduled with periodic tasks [13][14][15]. In these systems, the purpose of slack time analysis is to improve the response time of aperiodic tasks or to increase their acceptance ratio. However, those approaches require high time overheads to determine the available slack time at a given time which makes them inappropriate in our context. The concept of allowance was introduced in [16] in the case of Fixed Priority/Highest Priority First (FP/HPF).

The Real-Time Java specification proposes the use of handlers to detect time exceeding when executing a task (CostOverRunHandler) and for deadline exceeding (DeadlineMissHandler) [17].

According to the RTSJ, a real time thread has four constructors. One of its most important parameters is the ReleaseParameters, which allows defining tasks’ basic parameters, finishing model and the two handlers that may be used to cope with deadline miss or execution overrun. The allowance is the duration that a faulty task may be used to cope with deadline miss or execution overrun. The allowance is the duration that a faulty task can be allowed to proceed without compromising the real-time constraints of all the tasks. A mechanism based on a cost overrun handler should be used to detect that the allowance has been consumed.

Unfortunately, the cost overrun handler is not always available; the solution is then to set timers with the maximum response time of a task computed with the hypothesis that the allowance is always consumed [19].

We introduce in [19] the Last Execution Time(LET) mechanism based on this worst case response time computation. This mechanism improves the performance achieved by the use of the allowance notion. Indeed, when a task won’t use its allowance (no execution overrun) then the resources left can be recovered by the other tasks. Hence, we give a faulty task more chance to complete, while respecting all the real time constraints.

Another solution is to use the allowance in budget manager. The budget value can be used by any faulty task.

3. Concepts and notations

In this paper, we consider a set \( \tau = [\tau_1, \ldots, \tau_n] \) of \( n \) sporadic tasks.

A Sporadic task \( \tau_i \) is defined by:

- \( C_i \), the supposed worst case execution time (WCET)
- \( T_i \), the minimum inter-arrival time (abusively called the period)
- \( D_i \), the deadline (a task released at time \( t \) must be executed by \( t + D_i \))

We consider in this paper, that for any task \( \tau_i \), \( D_i \leq T_i \). Hence, if all tasks deadlines are met then there can be only one instance of any task pending. We now recall classical results in the uniprocessor context for real-time scheduling.

- A task is said to be non-concrete if its request time is not known in advance. In this paper, we only consider non concrete request times, as the activation request times are supposed to be unpredictable.
- Time is assumed to be discrete (task arrivals occur and task executions begin and terminate at clock ticks; the parameters used are expressed as multiples of the clock tick); in [18], it is shown that there is no loss of generality with respect to feasibility results through restricting the schedules to being discrete, once the task parameters are assumed to be integers (multiples of the clock tick) i.e. a discrete schedule exists, if and only if a continuous schedule exists.
- An idle time \( t \) is defined on a processor as a time, so that there are no tasks released before time \( t \) pending at time \( t \). An interval of successive idle times is called an idle period.
- A busy period is defined as a time interval \( [a,b) \), so that there is no idle time in \( [a,b) \) (the processor is fully busy) and so that both \( a \) and \( b \) are idle times. The synchronous busy period is the busy period resulting from the scenario where all tasks are all first released at time 0.
- FP/HPF denotes the preemptive Fixed Priority Highest Priority First algorithm with an arbitrary priority assignment.
- For any task \( \tau_i \), \( hp(i) \) denotes the set of tasks having a higher or equal priority than \( \tau_i \) but \( \tau_i \).
- \( U = \sum_{i=1}^{n} \frac{C_i}{T_i} \) is the processor utilization factor, i.e., the fraction of processor time spent in the execution of the task set [1]. An obvious necessary condition for the feasibility of any task set is \( U \leq 1 \) (this is assumed in the sequel).
The parameter values of the tasks are as follows:

- The tasks are at their maximum rate and released synchronously at time \( t=0 \). This corresponds to the worst case scenario for the feasibility of the tasks [7].
- The execution is carried out in a scenario in which all tasks are at their maximum rate and released synchronously time \( t=0 \).
- The execution is carried out in a scenario in which all tasks are at their maximum rate and released synchronously time \( t=0 \). This corresponds to the worst case scenario for the feasibility of the tasks [7].
- The tasks have decreasing priorities.
- The parameter values of the tasks are as follows:

<table>
<thead>
<tr>
<th>Task</th>
<th>( C_i )</th>
<th>( D_i )</th>
<th>( T_i )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \tau_1 )</td>
<td>4</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>( \tau_2 )</td>
<td>2</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>( \tau_3 )</td>
<td>3</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

Let us suppose that none of the three tasks exceeded its execution time. According to figure 1, representing the execution of the three tasks, we can notice the availability of many resources (time units) resulting from the idle period of the CPU.

Those idle times could be used by faulty tasks. We want to determine the maximum allowance (taken from idle period) that can be given to any faulty task with the assumption of \( k/W \) faults (with \( 1 \leq k \leq n \)).

Figure 1: allowance principles - example

We now show how to determine the task allowance for any fault model \( k/W \). The following recall the feasibility condition of a sporadic task set with \( k=0 \) fault.

**Theorem 1** [7]: The worst-case response time \( r_i \) of task \( \tau_i \) of a non-concrete periodic, or sporadic, task set (with \( D_i \leq T_i \) \( \forall i \in \{1,n\} \)) is found in a scenario in which all tasks are at their maximum rate and released synchronously at time \( t=0 \).

\( r_i \) is the solution of the following equation:

\[
r_i = C_i + \sum_{j \in hp(i)} \left( \frac{r_j}{T_j} \right) \times C_j
\]

The recursion ends when \( r^{m+1} = r^m \) and can be solved by successive iterations starting from \( r^0 = C_i \). It can be shown that this equation is always convergent if \( U \leq 1 \). A NSC is then \( U \leq 1 \) and \( \forall i \in \{1,n\}, r_i \leq D_i \).

We now suppose a fault model \( k/W \) and want to determine \( X_k \) the maximum allowance that can be given to any faulty task \( \tau_i \). We suppose that the allowance is fairly distributed among the faulty tasks.

\( \max(i,k) \) denotes the set of \( k-1 \) tasks (if any) in \( hp(i) \) having the smallest period.

**Theorem 2**: Suppose a fault model \( k/W \). The maximum allowance \( X_k \) for any faulty task \( \tau_i \) is the maximum value of \( X_k \) such that \( r_i \leq D_i \), where \( r_i \) is the solution of solution of following equation:

\[
r_i = C_i + X_k + \sum_{j \in hp(i)} \left( \frac{r_j}{T_j} \right) \times C_j + \sum_{j \in \max(i,k)} \left( \frac{r_j}{T_j} \right) \times (C_j + X_k)
\]

Proof:

Let \( \tau_i \) be a faulty task. By assumption, the maximum number of faulty tasks is \( k \). In the interval \( [0,t] \), the maximum execution time of any faulty task \( t_j \) is \( C_j + X_k \).

For any task \( t_i \), the worst case is that the \( k-1 \) faulty tasks have the smallest period.

5. **Budget management**

The allowance enables to compute the available resources which we can give to a task exceeding its WCET.

Instead of fixing the allowance for each task [16], we propose to create a new task in charge of the allowance management.

Two techniques are studied: the polling allowance and deferrable allowance.

5.1. **Polling Allowance**

In the "polling Allowance" technique, the faults correction is done by a sporadic task with the highest priority. The sporadic task is activated first after the execution of the first activation of the task with the highest priority among the task activated. The task is then periodic as long as the system is not idle. The task is suspended if the system becomes idle (no more task pending) until the next busy period. We propose in this paper to set the polling allowance (budget) duration and the polling period to the one corresponding to the periodic task with the shortest period.
The budget is only available at the polling task activation and is replenished every period in a busy period. The following table gives the values of allowance for the tasks set example:

<table>
<thead>
<tr>
<th>Task</th>
<th>Ci</th>
<th>Di</th>
<th>Ti</th>
<th>Ai</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ₁</td>
<td>4</td>
<td>10</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>τ₂</td>
<td>2</td>
<td>16</td>
<td>16</td>
<td>5</td>
</tr>
<tr>
<td>τ₃</td>
<td>3</td>
<td>20</td>
<td>20</td>
<td>6</td>
</tr>
</tbody>
</table>

**TABLE 1: Allowance computation**

In this case, the polling allowance task has as parameters: $C_{pa} = A_1$ ($τ_1$ is the most frequent task). $T_{pa} = T_1$.

The figure 2 shows an example of polling allowance task run first after the end of task $τ_1$ and the periodically.

The figure 2 shows an example of polling allowance task run first after the end of task $τ_1$ and the periodically.

**6. Simulations**

**6.1. Experimentation description**

In this experimentation, we make a comparison between three solutions which we propose to deal with execution overruns. These solutions are: “allowance use”, “Latest Execution Time” and “Deferrable allowance”. We recall that, these three solutions are studied for FP/HPF scheduling. The last execution time represents the last instant that we can execute a task without compromising the real-time constraints of the tasks [19].

The faults generation module uses a “Normal law”. The execution duration of a task $τ_i$ is given during each execution period, and the fault is detected when the task duration exceeds its execution duration $C_i$.

We consider a set of four tasks $τ=(τ_1,τ_2,τ_3,τ_4)$ with a processor utilization $U=0.7$.

Tasks $τ_1,…τ_4$ have decreasing priorities and the task parameters are as follows:

<table>
<thead>
<tr>
<th>Task</th>
<th>Ci</th>
<th>Di</th>
<th>Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td>τ₁</td>
<td>3</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>τ₂</td>
<td>2</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>τ₃</td>
<td>2</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>τ₄</td>
<td>5</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>

**TABLE 2: Tasks set - example**

We assume that every task can cause faults, which is made possible by modifying the task execution time according to the “Normal law” during each period. We will use the Box-Muller method: if $x_1$ and $x_2$ are two uniform random numbers on ]0,1[, the $y$ number is defined by:

$$y = \sqrt{-2 \ln(x_1) \cos(2\pi x_2)}$$

follows a reduced “Normal law”.

We deduce from this that the number: $z = m + (s \times y)$ and follow the normal law $N(m, s)$. Where : $m$ is the expected execution duration; and $σ$ the standard deviation.

In our simulation, we suppose that the execution duration overrun is at most equal to 50% of task execution time. So the average value $m$ will be equal to the task execution time (duration) and the standard deviation $s$ will be equal to the half of the faulty task execution time ($z = \ln(C_i/C_i/2)$).

During execution, $C_i$ value changes according to the normal law, we will take only the values higher or equal to $C_i$:

$$\begin{align*}
C_i^* &= z & \text{if } (z > C_i) \\
C_i^* &= C_i & \text{else}
\end{align*}$$
6.2. Experimentation results

In this experimentation, the faults number over a “sliding window” vary between 0 and n=4 (equal to the number of task.).

We consider the synchronous scenario where tasks are all first released at time 0. We study the scheduling obtained during a duration equal to a multiple of the LCM of the tasks. We suppose that each task can be faulty.

With each test we compute the total number of faults of all tasks as well as the failures number made by the three solutions enumerated before (y-axis).

The following figure shows that the failures are significantly reduced in all proposed solutions.

![Failures comparison](image)

**Figure 4 : Failures comparison**

We notice that the curve representing the “Deferrable Allowance” is (i) smaller than the one representing the “Allowance uses”, and (ii) is almost identical to that which represents the “LET”. According to our simulation results, we can say that the use of the Deferrable allowance is an interesting solution to prevent failures due to execution overruns.

![Deferrable vs. Polling Allowance](image)

**Figure 5 : Deferrable vs. Polling Allowance**

In the second test, we compare the following policies: the Deferrable Allowance and the Polling Allowance. Look figure 5:

The values represented on the x-axes indicate the number of tasks activations during the test.

In this test we compute the total number of failures produce by studied policies represented on the y-axis. The figure 5 shows that the Deferrable Allowance policy is more effective than Polling Allowance policy. The use of the Deferrable Allowance requires more resources.

7. Conclusion

In this paper we have considered the problem of the real-time systems conception when the task’s execution durations cannot be accurately determined.

We have studied different mechanisms for deadline miss prevention in real-time systems. Based on the concept of allowance, we have introduced the polling allowance and deferrable allowance solutions. The allowance use enables a faulty task to proceed with its execution without compromising the deadlines of others tasks. We have shown how to compute the allowance of a task with FP/HPF scheduling. We have shown in our experiments that the deferrable allowance solution is interesting to limit the impact of execution overrun faults and to increase the fault correction rate.

References


