A Probabilistic Approach for Fault Tolerant Multiprocessor Real-time Scheduling

Vandy Berten[1,*], Joël Goossens[1], Emmanuel Jeannot[2]

[1]Université Libre de Bruxelles
CP 212
Av. F.D. Roosevelt, 50
1050 Bruxelles – Belgium
{vandy.berten, joel.goossens}@ulb.ac.be

[2]Loria INRIA-Lorraine
Campus scientifique
54506 Vandœuvre les Nancy
France
emmanuel.jeannot@loria.fr

Abstract

In this paper we tackle the problem of scheduling a periodic real-time system on identical multiprocessor platforms, moreover the tasks considered may fail with a given probability. For each task we compute its duplication rate in order to (1) given a maximum tolerated probability of failure, minimize the size of the platform such at least one replica of each job meets its deadline (and does not fail) using a variant of EDF *namely* $\mathsf{EDF}^{(k)}$ *or (2) given the size of the platform, achieve the best possible reliability with the same constraints. Thanks to our probabilistic approach, no assumption is made on the number of failures which can occur. We propose several approaches to duplicate tasks and we show that we are able to find solutions always very close to the optimal one.*

1. Introduction

The use of computers to control safety-critical real-time functions has increased rapidly over the past few years. As a consequence, real-time systems — computer systems where the correctness of a computation is dependent on both the logical results of the computation and the time at which these results are produced — have become the focus of much study. Since the concept of "time" is of such importance in real-time application systems, and since these systems typically involve the sharing of one or more resources among various contending processes, the concept of scheduling is integral to real-time system design and analysis. Scheduling theory as it pertains to a finite set of requests for resources is a well-researched topic. However, requests in real-time environment are often of a recurring nature. Such systems are typically modelled as finite collections of simple, highly repetitive tasks, each of which generates jobs in a very predictable manner. These jobs have upper bounds upon their worst-case execution requirements, and associated deadlines. In this work, we consider periodic task systems, where each task makes a resource request at regular periodic intervals and we consider *fault tolerant systems*, since a job can fail due to either transient fault of a processor or internal error. We propose a technique, based on *task replication*, in order that all jobs of each task meet their deadline (in terms of probability), i.e., at least one replica of each task job meets its deadline and does not fail. Task replication requires either more powerful processor[s] or additional processors (in order to schedule the system). We choose the second approach here and we consider the scheduling on identical multiprocessor platforms. Thus, the tasks are scheduled on an identical multiprocessor platform using global EDF [6]. More precisely, we shall consider the scheduling algorithm $\mathsf{EDF}^{(k)}$ (see [4]) in order to reduce significantly the number of processors needed.

Reliability in hard-real time systems has been extensively studied in the literature. Two kinds of faults have to be distinguished. A fault is called *transient* when the processor on which the fault has occurred can be considered for further scheduling. A fault is said *fail-stop*, when the processor on which it has happened cannot be used later on.

After the detection of a fault, the task stopped by the fault can be dynamically restarted [3, 5]. There are several drawbacks of this approach. First, there

*Vandy BERTEN is Research Fellow for FNRS (Fonds National de la Recherche Scientifique), Belgium

1-4244-0054-6/06/$20.00 ©2006 IEEE

is a system overhead as it takes time to detect fault, determine which task has been stopped and reschedule the task. Second, the execution of the recovery task may affect the predictability of the system.

In order to address this issue, static duplication have been proposed. Each task is executed more than once in order to increase the probability that at least one copy meets its deadline (and does not fail). In [8], periodic and aperiodic tasks are scheduled. The reliability of the system is ensured in case of transient fault by duplicating the number of processors and executing backup copies on the duplicated processors. The main problem of this approach is that the number of processors required to schedule both primary and backup copies is increased.

In order to decrease the number of processors, [10] proposes that each task has only two copies: a primary (active) backup which is always executed and a secondary (passive) backup copy. When the primary copy finishes, the backup copy is forced to terminate in order to save resources. Hybrid approach have also been proposed in order to reduce the number of required processors. For instance in [2, 7] the backup copy is only executed in case of failure. They consider the fail-stop model and in case of failure the tasks are rescheduled to the remaining processors.

In order to improve the reliability without increasing too much the number of processors, probabilistic approaches have been recently proposed. In probabilistic approaches hardware or software components are characterized by a probability of failure. This probability can depend on the task duration, task complexity, external sensors, etc. Then, given an overall reliability (or tolerated probability of failure) a precise analysis can determine for each task if duplication is required. For instance in [1] each processor and communication link is associated with a failure rate. The authors then tackle the problem of scheduling a task graph with deadline constraints and guarantying the best possible reliability. An other advantage of the probabilistic approach is that, contrary to static or hybrid approach, no assumption on the number of tolerated failures is made.

In this paper, we discuss how to apply the probabilistic approach for scheduling a set of periodic real-time task. We shall study two symmetric problems. First, given a target reliability (i.e., a probability) find the smallest possible number of processors that achieves this reliability. Second, given the number of processors find the best achievable reliability.

The paper is organized as follows, in Section 2, we introduce our model of computation and our reliability problem. In Section 3, we study the two symmetric reliability problems. In Section 4, we present 5 heuristics for increasing the number of task copies. In Section 5, we present our simulation results in order to evaluate our heuristics. In Section 6, we conclude.

2. Background

2.1. Model of computation

We consider the scheduling of periodic hard real-time tasks, in the popular periodic task model ; a *task* $\tau_i = (C_i, T_i)$ is characterized by two parameters – an execution requirement C_i and a period T_i – with the interpretation that the task generates a *job* at each integer multiple of T_i, and each such job has an execution requirement of C_i execution units, and must complete by a deadline equal to the next integer multiple of T_i. We assume that preemption is permitted – an executing job may be interrupted, and its execution resumed later, with no loss or penalty. Moreover, we assume that T_i and C_i are natural integer, representing for instance a number of CPU tics. A periodic task system consists of several independent such periodic tasks that are to be executed on a specified preemptive processor architecture. Let $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ denote a periodic task system. For each task τ_i, we define its *utilization* U_i to be the ratio of τ_i's execution requirement to its period: $U_i \stackrel{\text{def}}{=} C_i/T_i$. We define the *utilization* $U_{\mathsf{sum}}(\tau)$ of periodic task system τ to be the sum of the utilizations of all tasks in τ: $U_{\mathsf{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} U_i$. Furthermore, we define the *maximum utilization* $U_{\mathsf{max}}(\tau)$ of periodic task system τ to be the largest utilization of any task in τ: $U_{\mathsf{max}}(\tau) \stackrel{\text{def}}{=} \max_{\tau_i \in \tau} U_i$. We shall also define the *hyperperiod* P as follows: $P \stackrel{\text{def}}{=} \text{lcm}\{T_i \mid \tau_i \in \tau\}$ and n_i the number of τ_i's requests in the interval $[O, P)$ (an integer) as follows: $n_i \stackrel{\text{def}}{=} \frac{P}{T_i}$. We shall also use the notion of *sporadic* task systems. *Sporadic* tasks are similar to periodic tasks, except that the parameters T_i denotes the *minimum* (rather than the exact) delay between successive jobs of τ_i. We assume that the switching times (which includes preemptions, migrations and scheduling) may be neglected (they are negligible in comparison with the task execution requirements.)

We consider in this work homogeneous multiprocessor platforms (all the processors are identical). The scheduling is preemptive and follows the global EDF [6] policy. "Global" scheduling algorithms, on the contrary to partitioned algorithms, allow different requests of the same task (also called jobs or processes) to be executed upon different processors. Each process can start its execution on any processor and may migrate

at run-time from one processor to another if it gets preempted by smaller-deadline processes.

We consider in this work *fault tolerant* systems. A task can fail due to either transient fault of a processor or internal error. In the case of a transient processor fault, the processor on which the fault has occurred can be considered for further scheduling [7]. Task internal error can be caused for instance by erroneous data transmitted from sensors and the processor on which the failure has occurred can also be considered for further scheduling. In any case, we assume that faults are not correlated. We allow *task's replication* and we request that at least one copy of each request does not fail and meets its deadline. More precisely, we extend the popular periodic task model and we consider, for each task τ_i, two additional parameters: p_i and t_i, where p_i is the probability of a task's request failure and t_i the number of copies of the task τ_i in the system. Consequently in our model each task is characterized by the 4-tuple (C_i, T_i, p_i, t_i) (note that in this more general case we have $U_{\mathsf{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} \frac{t_i C_i}{T_i}$).

For each task, the value p_i is supposed given by the user. For instance, in the case of transient fault occurring according to the exponential law with μ being the mean time between two failures, we have $p_i = 1 - e^{-\frac{C_i}{\mu}}$ which does only depend on the task τ_i.

2.2. Schedulability

We show here how to compute the size of a platform in order that all the tasks can be scheduled and meet their deadline (i.e., all replica of each task job meet their deadline if they do not fail).

We shall use the following result from [4], which relates schedulability (feasibility) upon (non-identical) multiprocessor platforms to EDF-feasibility upon identical multiprocessors.

Theorem 1 (Theorem 5 from [4]) Let I denote a hard-real time instance of jobs generated by the sporadic task system τ, which is feasible on a multiprocessor platform with total computing capacity $U_{\mathsf{sum}}(\tau)$ in which the fastest processor has a computing capacity $U_{\mathsf{max}}(\tau)$. Instance I is scheduled to always meet all deadlines on m processors each of computing capacity s by EDF, provided

$$U_{\mathsf{sum}}(\tau) \leq m \cdot s - (m-1) U_{\mathsf{max}}(\tau) \quad (1)$$

We consider in this work the scheduling of periodic tasks, the instance I is generated by a periodic task system τ, moreover we know that τ is feasible upon a (non-identical) multiprocessor platform with the total computing capacity be at least $U_{\mathsf{sum}}(\tau)$, and the fastest processor be of speed at least $U_{\mathsf{max}}(\tau)$. Notice that in our model of computation, the task execution requirements C_i consider unit-capacity processors ($s = 1$).

Consequently, from Equation 1, we can derive an expression for the minimum number of processors: $m \geq \left\lceil \frac{U_{\mathsf{sum}}(\tau) - U_{\mathsf{max}}(\tau)}{1 - U_{\mathsf{max}}(\tau)} \right\rceil$

Since t_i is the number of τ_i replicas, we get:

$$m \geq \left\lceil \frac{\sum_{\tau_i \in \tau} \frac{t_i \cdot C_i}{T_i} - U_{\mathsf{max}}(\tau)}{1 - U_{\mathsf{max}}(\tau)} \right\rceil \quad (2)$$

Algorithm 1 Computing the platform size

function **size**(τ,n)
Input: τ a system of n periodic tasks sorted by decreasing $\frac{C_i}{T_i}$
Output: m a sufficient number of processors to schedule all the tasks
return $\min_{k \in [1,n]} \left((\sum_{\ell=1}^{k-1} t_\ell) + \left\lceil \frac{U_{\mathsf{sum}}(\tau^k) - U_{\mathsf{max}}(\tau^k)}{1 - U_{\mathsf{max}}(\tau^k)} \right\rceil \right)$

Equation 2 gives a sufficient condition on the number of processors. However, if for some tasks $C_i \approxeq T_i$ we have $U_{\mathsf{max}} \approxeq 1$ which leads to an infinite value for m. Hopefully this problem can be overcame using the $\mathsf{EDF}^{(k)}$ scheduling policy which greatly reduces the number of required processors. Without loss of generality let us assume that the tasks are sorted by decreasing value of $\frac{C_i}{T_i}$. Then compute $m_{\min}(\tau) = \min_{k \in [1,n]} (\sum_{\ell=1}^{k-1} t_\ell) + \left\lceil \frac{U_{\mathsf{sum}}(\tau^k) - U_{\mathsf{max}}(\tau^k)}{1 - U_{\mathsf{max}}(\tau^k)} \right\rceil$ where $\tau^k \stackrel{\text{def}}{=} \{\tau_k, \tau_{k+1}, \ldots, \tau_n\}$ is the system τ restricted to the $n-k+1$ last tasks (the $n-k+1$ tasks that have the lowest value of $\frac{C_i}{T_i}$). One can check that the system can be scheduled with $m_{\min}(\tau)$ processors. Indeed for each value of k, the k first tasks can be schedule on $\sum_{\ell=1}^{k-1} t_\ell$ processors with EDF once each of them is assigned $-\infty$ as (virtual) deadline while the remaining $n-k$ tasks can be schedule with EDF using the number of processors given by Equation 2. The procedure **size** in Algorithm 1 summarizes this approach.

2.3. Reliability

Let us first characterize the probability to meet the deadline (and not fail) of at least one replica of each task's request in the interval $[0, P)$.

Theorem 2

$$\mathcal{P} = \prod_{i=1}^{n} (1 - p_i^{t_i})^{n_i},$$

where $\mathcal{P}$ is the probability that all tasks meet their deadline in the interval $[0, P)$, i.e., at least one copy of each task's request meets its deadline and does not fail in $[0, P)$.

Proof. Please notice that we assume that faults are independent (not correlated) and that the processor on which the fault has occurred can be used for further scheduling. First we characterize the probability $\mathcal{P}_i^1$ that a single request of the task τ_i successes, which is $1 - q_i$, where q_i is the probability that all the copies of the request of τ_i fail. Hence, $q_i = \prod_{j=1}^{t_i}(p_i)$, hence $\mathcal{P}_i^1 = 1 - p_i^{t_i}$.

Now we characterize the probability $\mathcal{P}_i^{n_i}$ that the n_i requests of τ_i success in the interval $[0, P)$, $\mathcal{P}_i^{n_i} = (1 - p_i^{t_i})^{n_i}$. Consequently, $\mathcal{P} = \prod_{i=1}^{n}(1 - p_i^{t_i})^{n_i}$. □

The probability $\mathcal{P}$ gives the probability of success during an hyper-period. This value makes the comparisons between different systems difficult. It would be natural that a system is "as reliable as" another one if, for a same period of time, the probability of failure is the same. This is why, users often express the probability of failure over a period of time: for instance in commercial flight-control system the required probability of failure have to be approximately 10^{-10}/hour [9].

Moreover, in real systems, hyper-period is often very large and leads to numerical issues. It is then convenient to use smaller observation periods.

Let us now consider the success probability of the system τ over a time frame F (large in comparison with the task periods), which we note $\mathcal{P}_F$. We have:

Theorem 3

$$\prod_{i=1}^{n}(1 - {p_i}^{t_i})^{\lceil \frac{F}{T_i} \rceil} \leq \mathcal{P}_F \leq \prod_{i=1}^{n}(1 - {p_i}^{t_i})^{\lfloor \frac{F}{T_i} \rfloor}$$

Proof. We will use a reasoning close to the Theorem 2. Let $\mathcal{P}_i^F$ be the success probability of all copies of task τ_i in the interval $[0, F)$. We can see that $\mathcal{P}_i^F$ can be splitted into two parts:

1. success probability of all copies of all requests of τ_i having their deadlines before F,

2. success probability of all copies of the request (if such a request exists) of τ_i starting before F and having its deadline after F.

The first part can be obtained in the same way as in Theorem 2, and is equal to $(1 - {p_i}^{t_i})^{\lfloor \frac{F}{T_i} \rfloor}$. We cannot compute exactly the second part if we do not know the error time distribution and the scheduling algorithm. But we know that this value is between 1 (if there is no such request, which is the case is $\lfloor \frac{F}{T_i} \rfloor = \frac{F}{T_i}$), and $1 - {p_i}^{t_i}$ if $\frac{F}{T_i}$ is just below $\lceil \frac{F}{T_i} \rceil$. We have then that

$$(1 - {p_i}^{t_i})^{\lceil \frac{F}{T_i} \rceil} \leq \mathcal{P}_i^F \leq (1 - {p_i}^{t_i})^{\lfloor \frac{F}{T_i} \rfloor}, \forall i$$

The property follows directly from this inequality. □

In the following, we will use the approximation:

$$\mathcal{P}_F \simeq \prod_{i=1}^{n}(1 - {p_i}^{t_i})^{\frac{F}{T_i}} \quad (3)$$

We justify this approximation in the following way: the difference (in product) between the two terms of the inequality is lower than $\prod_{i=1}^{n}(1 - {p_i}^{t_i})$ (and is equal to this value if $\lceil \frac{F}{T_i} \rceil \neq \frac{F}{T_i} \forall i$). This value can be neglected if the ratio $\frac{F}{T_i}$ is large in comparison to 1. Our approximation is a kind of interpolation (or an average) between the two bounds of our range.

In conclusion, we can choose a time frame F, and ϵ the maximum tolerated probability of failure on $[0, F)$, and check that $1 - \mathcal{P}_F \simeq 1 - \prod_{i=1}^{n}(1 - {p_i}^{t_i})^{\frac{F}{T_i}} \leq \epsilon$ or check that $1 - \prod_{i=1}^{n}(1 - {p_i}^{t_i})^{\lceil \frac{F}{T_i} \rceil} \leq \epsilon$ to be sure that $1 - \mathcal{P}_F \leq \epsilon$ since $\prod_{i=1}^{n}(1 - {p_i}^{t_i})^{\frac{F}{T_i}}$ is only an approximation.

3. Algorithms

In this section we tackle two symmetric problems. First, given a target reliability, find the smallest possible number of processors that achieves this reliability. Second, given the number of processors, find the best achievable reliability. For both problems we assume that the periodic task system is given and that tasks are sorted by decreasing value of $\frac{C_i}{T_i}$. Therefore we have to find t_i the number of copies of each task.

3.1. Minimizing the number of processors

Algorithm 2 Minimizing the platform size

Input: τ: a sorted system of periodic tasks
  ϵ: maximum tolerated probability of failure
  F: a time frame of study
Output: m the size of the platform
  $\forall i\, t_i$ the number of copies of task τ_i
1 $\forall i\, t_i \leftarrow 1$
2 **while** $1 - \prod_{i=1}^{n}(1 - p_i^{t_i})^{\frac{F}{T_i}} > \epsilon$ **do**
3   increase copies of some tasks according to one of the heuristics of Section 4.
4 $m \leftarrow$**size**(τ, n)

For minimizing the size of a platform we propose the Algorithm 2. In entry it takes, a periodic system τ, ϵ a maximum tolerated probability of failure and F a time frame. It outputs t_i the number of copies of each tasks τ_i and m the size of the platform such as the following constraints are met: (1) m is small as possible, (2) in the interval $[0, F)$ the probability of failure of the proposed solution is approximately lower than ϵ, and (3) the task set is EDF-schedulable on m unit-capacity processor (if we do not consider job failures).

At the beginning of the Algorithm 2 the number of copies of every tasks is set to 1. Then, while $1-\mathcal{P}_F > \epsilon$, we increase the number of copies of the tasks until the probability of failure of the system is smaller than ϵ. Then using Algorithm 1, we compute a sufficient number of processors such that the system is schedulable.

In Section 4 we will discuss the different ways to increase the number of copies of the tasks.

3.2. Optimizing reliability

Algorithm 3 Minimizing the failure probability

Input: τ: a sorted system of periodic tasks
  m: the size of the platform
  F: the time frame of study
Output: ϵ: maximum tolerated probability of failure
  $\forall i\, t_i$ the number of copies of task τ_i
1 $\forall i\, t_i \leftarrow 1$
2 **while size**(τ,n)$< m$ **do**
3   increase copies of some tasks according
    to one of the heuristics of Section 4.
4 cancel the last increase of copies
5 $\epsilon \leftarrow 1 - \prod_{i=1}^{n}(1-p_i^{t_i})^{\frac{F}{T_i}}$

For minimizing the probability of failure we propose the Algorithm 3. It takes, a periodic system τ, m the size of the platform and F the time frame of study. The output is the number of copies of each task and ϵ the probability of failure in $[0, F)$. It meets the following constraints: (1) ϵ is as small as possible, (2) the required size of the platform is not greater than m and (3) the task set is EDF-schedulable on m unit-capacity processor (if we do not consider job failures).

At the beginning of the Algorithm 3 the number of copies of all the tasks is set to 1. Then as long as the system is schedulable, we increase the number of copies of the tasks. Schedulability is checked using Algorithm 1. Then we compute the probability of failure of the system using eq. (3).

Note that again, there is several ways to increase the number of copies of the tasks (see next section).

4. Duplicating tasks

By definition of $\mathcal{P}_F$ we know that $\mathcal{P}_F$ increases as t_i increases. This means that duplicating tasks increases the probability that at least one copy of each task job meets its deadline and does not fail, or symmetrically, that the probability of failure decreases.

Moreover, we see that from Equation 2 increasing t_i may lead to an increase of m. This means that duplicating tasks increases the required size of the platform.

Therefore, in order to solve our problem, we need to carefully choose which task to duplicate. Moreover the way we choose how to duplicate the task will influence the speed of convergence of each algorithm. The convergence of each algorithm is guaranteed by ensuring that any task can potentially be duplicated infinitely.

We propose 5 heuristics for choosing how to increase task's number of copies.

Increase all The simplest solution consists in increasing the number of copies of all the tasks:

$$\forall i\, t_i \leftarrow t_i + 1.$$

We do not expect that this heuristic will give good results in general but it will be a very fast one.

Min utilization In Equation 2 the number of processors increases with $U_{\mathsf{sum}} = \sum_{\tau_i \in \tau} \frac{t_i C_i}{T_i}$. With this heuristic we increase t_i for the task where

$$i = \text{argmin}\left(\frac{t_i C_i}{T_i}\right).$$

This heuristic aims at choosing the task that, once duplicated, will increase minimally the number of processors. The drawback is that it does not take into account the probability of failure of tasks.

Min failure In this heuristic we duplicate the task that have the highest probability to fail: increase t_i such that

$$i = \text{argmax}\left(p_i^{t_i}\right).$$

This duplication scheme is only based on probability and not on utilization.

Min failure-request In this heuristic we duplicate the task for which the probability to fail times its number of requests is the highest: increase t_i such that

$$i = \text{argmax}\left(\frac{F}{T_i} p_i^{t_i}\right).$$

The idea is to duplicate tasks with the largest probability such that at least one of its request fails in $[0, F)$.

Min failure-utilization The idea of this heuristic is to duplicate tasks that have a high probability of failure and low utilization: increase t_i such that

$$i = \text{argmin}\left(\frac{C_i}{T_i p_i^{t_i}}\right).$$

The idea is to duplicate tasks that have a high failure probability. As the previous heuristic, this one aims at taking into account all the tasks characteristics.

5. Simulation results

We have implemented the 5 variants of each proposed algorithm (10 heuristics in total). In order to evaluate each heuristic we have generated several scenarios composed of a random periodic systems and a random value of ϵ (in the case of minimizing the number of machines) or m (in the case of optimizing the reliability). We have used the following parameters: $N_{\max}$ the maximum number of tasks, $T_{\max}$ the maximum period and F the time frame of study (*i.e.* the duration during which we compute the probability of failure according to Eq. (3). We denote by $U[a, b]$ a random number chosen uniformly in $[a, b]$. For building a periodic system we first compute the number of tasks in $U[1, N_{\max}]$. Then for each task τ_i we compute its period T_i in $U[1, T_{\max}]$. Its duration C_i is randomly chosen in $U[1, T_i]$. Finally, in order to have a fine control of the heterogeneity of the system, we use three classes of probability of failure. We pick up a class c in $U[1, 3]$ and choose p_i in $U[10^{-12}, 10^{-10}]$ for the first class, in $U[10^{-8}, 10^{-6}]$ for the second class, or in $U[10^{-2}, 10^{-1}]$ for the third class. In the following experiments are done on a powerpc G4 at 1GHz with 1GB of RAM running MacOSX 10.4.4 and we will have: $N_{\max} = 30$, $T_{\max} = 50$ and $F = 360000$ (if a unit of time is 10 ms, then F is 1 hour).

5.1. Minimizing the number of processors

In Table 1, we compare the heuristics over 1000 scenarios for 3 classes of tasks and ϵ is chosen in $U[10^{-8}, 10^{-6}]$. Column nb_{win} displays the number of times the heuristics have given the best results (against the other heuristics), column m_{sum} gives the sum of all the platform sizes computed by the heuristics and column time displays the cumulative run-time of each heuristic. Table 2 displays the same data but for only one class of task (the first one).

Table 1 Comparing heuristics for the platform minimization problem (3 classes of tasks)

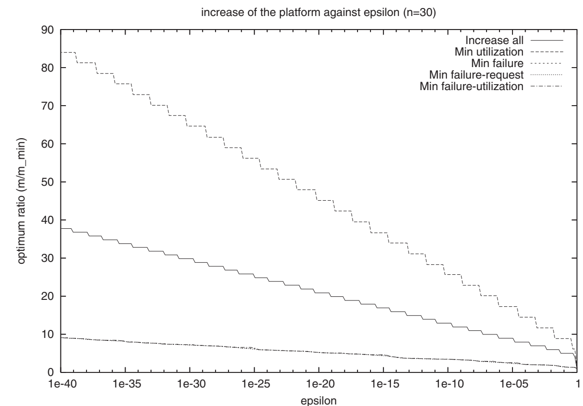
Heuristic	nb_{win}	m_{sum}	time (s)
Increase all	65	139273	0.219
Min utilization	60	240388	11.002
Min failure	607	58260	1.416
Min failure-request	972	57544	1.466
Min failure-utilization	554	58529	1.480

Table 2 Comparing heuristics for the platform minimization problem (1 class of tasks)

Heuristic	nb_{win}	m_{sum}	time (s)
Increase all	189	26664	0.0515
Min utilization	83	39491	1.200
Min failure	670	25112	0.353
Min failure-request	972	24698	0.331
Min failure-utilization	599	25250	0.351

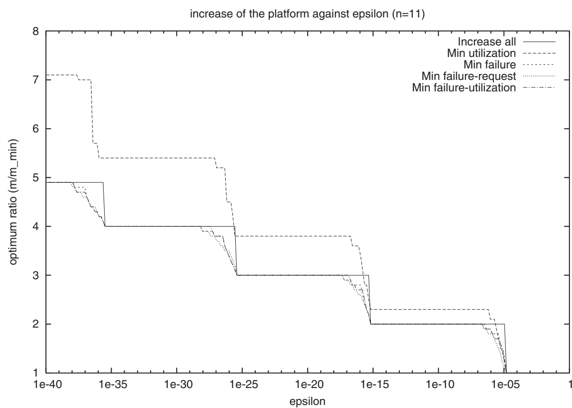
From these tables we see that the heuristics which do not take into account probability are less efficient that the others. We see that the *Increase all* heuristic is the fastest one, and that the heuristics which take into account probability (the last three one) give similar results, *Min failure-request* being the best of the 3. We also see that *Min utilization* is the slowest heuristic. The main reason is that, in many cases, it does not choose the right task to duplicate and therefore it takes a longer time to find a duplication scheme that meets the reliability requirements.

Figure 1 epsilon vs. $m/m_{\min}$ (3 classes of tasks)



In Figure 1 and 2 we plot how varies the ratio $m/m_{\min}$, where m is the solution of a heuristic and $m_{\min}$ is the minimum size of the platform (when $\epsilon = 1$

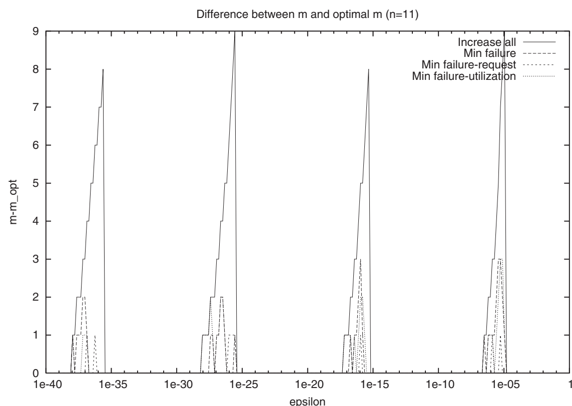
Figure 2 epsilon vs. $m/m_{\min}$ (1 class of tasks)



and no task is duplicated). We have $n = 30$ for Fig. 1 and $n = 11$ for Fig. 2.

These figures corroborate the previous results. It shows that the 3 heuristics *Min failure*, *Min failure-request*, *Min failure-utilization* have about the same performance, *Min failure-request* being slightly the best one.

Figure 3 epsilon vs. $m - m_{\text{opt}}$



Moreover we see that it is required to have about a platform 10 times larger than the case $\epsilon = 1$ to have a failure probability of 10^{-40} for the 3 classes of task problem. When the probability of failure of the task is small enough, for $\epsilon = 10^{-40}$, we see that the reliability is ensured by having a platform 5 times larger than the original one. This number might seem large. In order to counter this intuition we plot in Figure 3, the difference $m - m_{\text{opt}}$, where m is the solution of a heuristic and m_{opt} is the optimal solution (found by exhaustive search). We see that, for our 3 best heuristics, the difference between the heuristic and the optimal is most of the time 0, moreover, it never exceeds 3 for these heuristics.

5.2. Optimizing Reliability

In Tables 3 and 4 we show the results of the comparison of our heuristics for the reliability optimization problem. For each experiment the target size of the platform is chosen in $U[m_{\min}, 3 \cdot m_{\min}]$ where $m_{\min}$ is the size of the platform assuming that no duplication is allowed. In column ϵ_{avg} we plot the geometric average of the found probability of failure given the target size.

Table 3 Comparing heuristics for the reliability optimization problem (3 classes of tasks)

Heuristic	nb_{win}	ϵ_{avg}	time (s)
Increase all	317	$6.228{\cdot}10^{-01}$	0.083
Min utilization	321	$6.066{\cdot}10^{-01}$	1.487
Min failure	574	$7.998{\cdot}10^{-02}$	0.942
Min failure-request	881	$7.028{\cdot}10^{-02}$	0.931
Min failure-utilization	598	$8.276{\cdot}10^{-02}$	1.021

Table 4 Comparing heuristics for the reliability optimization problem ($p_i \in U[10^{-12}, 10^{-10}]$)

Heuristic	nb_{win}	ϵ_{avg}	time (s)
Increase all	71	$1.260{\cdot}10^{-10}$	0.094
Min utilization	71	$7.582{\cdot}10^{-07}$	1.906
Min failure	282	$1.943{\cdot}10^{-11}$	1.030
Min failure-request	863	$1.271{\cdot}10^{-11}$	1.047
Min failure-utilization	318	$2.195{\cdot}10^{-11}$	1.121

We see that once again, *Min failure*, *Min failure-request*, *Min failure-utilization* provide similar performances *Min failure-request* being the best one. We also see that for this problem *Min utilization* is the slowest heuristic while *Increase all* is fastest one.

In Fig. 4 (resp. 5) we plot the evolution of the computed reliability against the $m/m_{\min}$ where m is the target size of the platform and $m_{\min}$ is the size of the platform with no duplication, for 30 (resp. 11) tasks system with 3 (resp. 1) classes of tasks. We see that (as the y axes is logarithmic) that the probability of failure decreases exponentially with the size of the platform. Once again we see that *Min failure*, *Min failure-request*, *Min failure-utilization* provide the best reliability for any size of the platform.

Figure 4 epsilon vs. $m/m_{\min}$ for 3 classes of tasks.

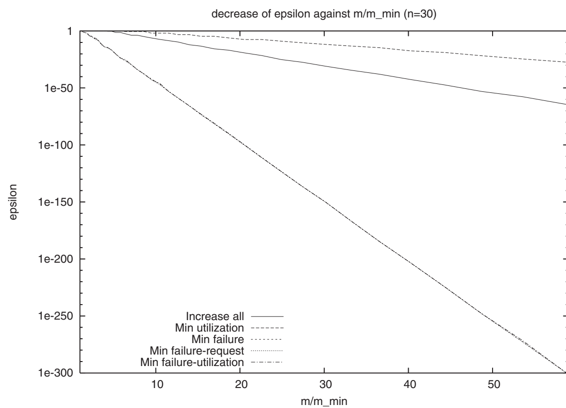
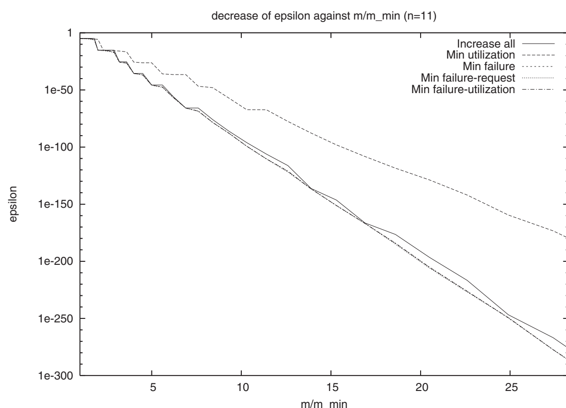


Figure 5 epsilon vs. $m/m_{\min}$ for 1 class of task.



6. Conclusion

In this paper, we considered the scheduling of hard real-time periodic task sets on identical multiprocessor platforms using $\mathsf{EDF}^{(k)}$. We considered *fault tolerant systems*, since jobs can fail due to either transient fault of a processor or internal error. We proposed a technique based on *task replication* in order that all jobs of each task meet their deadline (in terms of probability), i.e., at least one replica of each task job meets its deadline and does not fail. We studied two symmetric problems. First, given a target reliability find the smallest possible number of processors that achieves this reliability. Second, given the number of processors find the best achievable reliability. We proposed first a technique, based on $\mathsf{EDF}^{(k)}$ [4], for minimizing the size of the multiprocessor platform (i.e., the number of processors) and based on that we proposed various heuristics for choosing the number of copies of each task. Simulation results showed that the heuristics based on the probability failure of task are significantly more efficient than the others. The experiments showed also that the heuristic *Min failure-request* is the best one. We also showed that our 3 best heuristics (based on the failure probability) are nearly optimal.

In further work we plan to tackle the problem where processor speed are different or where probability of failure depends on the processors. For the first case this requires to extend our schedulability test, while for the second case a new characterization of the reliability has to be found.

References

[1] I. Assayad, A. Girault, and H. Kalla. A Bi-Criteria Scheduling Heuristics for Distributed Embedded Systems under Reliability and Real-Time Constraints. In *Proc. Intl. Conf. on Dependable Sys. and Net. (DSN'04)*, Firenze, Italy, June 2004.

[2] A. A. Bertossi, A. Fusiello, and L. V. Mancini. Fault-Tolerant Deadline-Monotonic Algorithm for Scheduling Hard-Real-Time Tasks. In *Proc. 11th Intl. Parallel Proc. Symp.*, 1997.

[3] S. Ghosh, R. Melhem, and D. Mossé. Fault-Tolerant Scheduling on a Hard Real-Time Multiprocessor System. In *Proc. 8th Intl. Parallel Proc. Symp.*, pages 775–782, 1994.

[4] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on uniform multiprocessors. *Real Time Systems*, 25:187–205, 2003.

[5] C. M. Krishna and K. G. Shin. On scheduling Tasks with a Quick Recovery from failure. *IEEE Trans. Computer*, C-35(5):448–455, 1986.

[6] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[7] G. Manimaran and C. Siva Ram Murthy. A Fault-Tolerant Dynamic Scheduling Algorithm for Multiprocessor Real-Time Systems and its Analysis. *IEEE Trans. on Parallel Dist. Syst.*, 9(11):1137–1152, Nov. 1998.

[8] Y. Oh and S. H. Son. An Algorithm for Real-Time Fault-Tolerant Scheduling in Multiprocessor Systems. In *Fourth Euromicro workshop on Real-Time Systems*, pages 190–195, 1992.

[9] S. Shatz, J. Wang, and M. Goto. Task Allocation for Maximizing Reliability of Distributed Computer Systems. *IEEE Trans. on Computers*, 41:156–168, Sept. 1992.

[10] T. Tsuchiya, Y. Kakuda, and T. Kikuno. A New Fault-Tolerant Scheduling Technique for Real-Time Multiprocessor Systems. In *Proc. 2nd International Workshop on Real-Time Computing Systems and Applications*, pages 197–202, 1995.