Dynamic Feasibility Window Reconfiguration for a Failure-Tolerant PFair Scheduling

Yves Mouafo Tchinda, Annie Choquet-Geniet, and Gaëlle Largeteau-Skapin

LIAS-ENSMA
Teleport2, 1 av. Clément Ader BP 40109
86961 Futuroscope-Chasseneuil France
XLIM-UNIVERSITE DE POITIERS
Teleport2, 11 bd Marie et Pierre Curie,BP 30179
86962 Futuroscope-Chasseneuil France
`yves.mouafo@ensma.fr,annie.geniet@univ-poitiers.fr,`
`gaelle.largeteau.skapin@univ-poitiers.fr`
`http://www.lias-lab.fr; http://www.xlim.fr/sic`

Abstract. This work addresses the problem of failure tolerance for real-time applications. We propose a technique to schedule a system of tasks running under PD2 on a multicore architecture submitted to the failure of one of the cores at a given time with re-execution of the task running on it. The technique is based on limited hardware redundancy and subtask feasibility windows reconfiguration at runtime. We limit the study to the re-execution of one time unit.

Keywords: Failure Tolerance, Multicore Architecture, Task Re-execution, Pfair Scheduling, Dynamic Reconfiguration

1 Introduction

In recent decades, there has been a significant increase in the use of multicore architectures in real-time embedded systems [1]. A multicore processor has two or more independent cores into a single package. The drawback of such an architecture is its weakness because a core may fail at any time requiring an adaptation of the system [2]. Therefore, designers of systems running on such architectures attach significant importance to the failure tolerance.

We consider systems of critical periodic tasks running on a multicore architecture. We use the classical temporal modelling of tasks. Each task τ_i is characterized by four temporal parameters: the first release date or offset r_i, the worst-case execution time (WCET) C_i, the period T_i and the relative deadline D_i which is the maximum delay allowed from the release of any instance of the task to its completion.
If all the first release dates are equal, the tasks are said to have simultaneous first releases. When the deadlines verify $D_i \leq T_i$, we say that they are constrained; when $D_i{=}T_i$, they are implicit. A system S is characterised by its processor's utilisation factor, which represents the processor workload due to the system.

It is defined by $U = \sum_{i=1}^{n}(C_i/T_i)$. Finally, $H = LCM(T_1...T_n)$ denotes the hyper-period of the system.

In the sequel, we consider a system composed of n periodic independant tasks with simultaneous first releases and implicit deadlines. A task τ_i is denoted by a triplet $< C_i, D_i, T_i >$ when the deadlines are constrained or by a pair $< C_i, T_i >$ when they are implicit. We consider here global scheduling with the algorithm PD2 (see 3.2), which is optimal in our context. In addition, there exists a necessary and sufficient feasibility condition easy to compute. The system is feasible on k cores iff $\sum_{i=1}^{n}(C_i/T_i) \leq k$ [5]. PD2 recommends to divide tasks into unitary subtasks. If one of the cores fails, it affects the subtasks running on it. In [3], we proved that if the affected subtasks are simply ignored, limited hardware redundancy (see 4.1), which consists of the addition of a single further core, provides a valid schedule (i.e a schedule which meets all its temporal constraints). In the case where the affected subtask must be re-executed, the main issue is to ensure the validity of the schedule, even if the re-execution causes delays.

We propose a technique based on a dynamic reconfiguration of the subtask feasibility windows at runtime. The execution starts with constrained feasibility windows. After the failure detection, the feasibility windows are released. The constrained windows are computed by means of a task system with constrained deadlines, using a method named ghost substask method (see 4.2).
We prove that, together with the limited hardware redundancy, this technique of contraction and relaxation of feasibility windows provides a valid schedule.

The remainder of this paper is organized as follows: in Section 2, we present the scheduling algorithm PD2 and the failure models of interest. Then, a state of the art is given (Section 3). It is followed by the presentation of our approach (section 4) and the priority inversion issue (Section 5). The paper is concluded with the presentation of experimental results and some elements of proof of our central feasibility result.

2 Context and problematic

2.1 Scheduling algorithm

PD2 [4] is a PFair (Proportionate Fair) algorithm. PFair algorihtms require the execution of tasks at a regular rate, the objective is to approach an ideal scheduling in which each task τ_i receives exactly $U_i \times t$ processor time units in the range $[0, t)$. The construction of a PFair scheduling requires to divide each task τ_i into unitary subtasks $\tau_i^j (j \geq 0)$. Each subtask has a pseudo-release date $r_i^j = \lfloor \frac{j}{U_i} \rfloor$ and a pseudo-deadline $d_i^j = \lceil \frac{j+1}{U_i} \rceil$, with $j \geq 0$ and $U_i = \frac{C_i}{T_i}$. The interval $[r_i^j, d_i^j)$ represents the feasibility window of the subtask τ_i^j. Scheduling a subtask τ_i^j in its feasibility window means that τ_i runs for one time unit within the time inverval $[r_i^j, d_i^j)$. Here the notion of subtask refers to a time division not to a software division.

PD2 is recognized as the most efficient Pfair algorithms because it has a low complexity for decision-making. A subtask has priority over another if it has the

smallest deadline. In the case of equality of deadlines, PD2 uses two additional criteria based on the bit succesor b_i^j and the group deadline D_i^j.
If τ_i^j and τ_i^{j+1} denote two successive subtasks of the task τ_i, $b_i^j = 1$ if their feasibility windows overlap and $b_i^j = 0$ otherwise.
If $U_i \geq 0.5$, $D_i^j = \lceil \frac{d_i^j - j - 1}{1 - U_i} \rceil$; otherwise $D_i^j = 0$ ($j \geq 0$).

Then, a subtask τ_i^j has priority over the subtask τ_q^k if
$(d_i^j < d_q^k)$ OR $(d_i^j = d_q^k \wedge b_i^j > b_q^k)$ OR $(d_i^j = d_q^k \wedge b_i^j = b_q^k = 1 \wedge D_i^j > D_q^k)$ OR $(d_i^j = d_q^k \wedge b_i^j = b_q^k = 1 \wedge D_i^j = D_q^k \wedge i < q)$. The last condition guarantees the determinism. In section 5 we present an example.

2.2 Failure modeling

All computer systems are subject to hardware and software failures. Hardware failures can be categorised into permanent, transient and intermittent failures [6]. Permanent failure, such as wear off of any part, requires the replacement of the spare part to restore the system functionality and does not disappear by itself with time. When a core cycles between being working and out of work, the failure is said intermittent. Transient failures are short term failures. They may occur because of external noise or temporal disturbance due to unidentified source; the core then recovers after a while. We consider here permanent failure. We assume that during the processing only one core fails. In this context there are two possible scenarios depending on the delay of detection:

1. Either the failure is detected instantly by means of a mechanism such as the ones presented in [7]. This means that no execution is lost and thus the problem of giving additionnal time to some tasks doesn't arise.
2. Or the failure is detected after x time units. In this case, we can consider three cases: (a) The current instances of the affected tasks must be fully executed: additional time is allocated to regain the lost execution and complete the tasks; (b) It is not mandatory to fully execute the current instances of the impacted tasks: there is thus no further time allocation. The affected tasks will use their remaining time to reexecute what has been lost and then continue execution until full use of the allocated time. This is for example the case for tasks which compute a result by means of successive iterations. A shorter execution means a less precise result, but doesn't lead to malfunctions; (c) The current instances of the affected tasks can be discarded. The scheduling continues with the unaffected task instances.

2.3 Outline of the study

Failure treatment consists of failure diagnosis and failure passivation [8]. Failure diagnosis consists of the location of the source of the failure, i.e the affected (hardware or software) component(s), and the determination of the nature of the failure (transient, permanent or intermittent). Passivation consists in providing an emergency response to the identified failure. It is our concern in this paper.

We focus on permanent failures. We consider that the diagnosis stage has been completed, that the failure is detected one time unit after its occurrence and the failing core identified. Thus, only one task is affected and we assume that its current instance must be fully completed. Therefore we are in the case $2-a$ of the previous description with $x = 1$. According to the task decomposition into unitary subtasks, only one substask of the affected task will have to be re-executed. Since the tasks are periodic with simultaneous first releases, the system is cyclic with period H. Thus, we limit the study to the first hyper-period $[0, H)$. Before presenting our solution based on limited hardware redundancy combined with feasibility windows reconfiguration, we first give an overview of existing related results in the litterature.

3 State of the art

3.1 Related works

The aim of failure tolerance is to schedule tasks in such a way that deadlines are still met despite processor or software failure. Several studies have been devoted to that issue. In this section, we are interested in those on tolerance to hardware failures on a multicore or multiprocessor architecture. The classical way to provide fault-tolerance on multicore platforms is to use redundancy [9]. The idea is to introduce redundant copies of the elements to be protected (processor or other components), and exploit them in the case of a fault. Therefore, hardware, time, information and software redundancies are used for fault-tolerance [10]. Of these types, time and hardware redundancy are the most frequently used. There are two main techniques for hardware redundancy: Triple Modular Redundancy (TMR) and Primary/Backup (PB). In TMR, three processors run redundant copies of the same workload and mask errors by voting on their outputs. In PB, two copies of each task are scheduled on different processors. The primary copy is executed and its output is checked for correctness by an acceptance test. If the acceptance test is negative, the execution of the backup copy is initiated.

While there are some variations from one approach to another, the general method to respond to a failure can be described as follows:

- *Transient failures*: If the system is designed only to withstand transients that disappear quickly, reexecution of the failed task or of a shorter, simpler version of that task, is carried out. The scheduling problem reduces to ensuring that there is always enough time to carry out such an execution before the deadline.
- *Permanent failure*: Backup versions of the tasks assigned to the failing core must be invoked. The steps are: provide each task with a backup copy; place the backups in the schedule; if the processor fails, activate one backup for each task that has been affected.

Most papers found in the litterature concern transient failures. The main techniques are given in [2]. In [6], authors propose an algorithm which is the

combination of TMR and DMR (Double Modular Redundancy) with an hybrid scheduling, based on the addition to the task parameters of an architectural vulnerability factor (AVF). [11] introduce checkpointing optimization. While running, each task records its states at checkpoints. When a failure occurs, the backup can resume execution from the lastest ckeckpoint.
Other authors address the use of PB technique for a fault tolerant EDF scheduling [12]. Notice that the papers cited above concern partioned scheduling. For global scheduling there is little work. We can mention [13] which proposes an algorithm (named FT-PF) for a tolerant Pfair Scheduling. This algorithm increases the utilization of a task to ensure that it completes execution V_i time units before its deadline. So, when a task must recover, it can be allocated V_i time units for recovery. FT-PF also uses a spare core to ensure that recovery will not fail for lack of resources. This technique is similar to what we propose for a permanent failure in the way that it modifies task temporal parameters and uses one more core. The difference is that, in our approach, this core is not kept idle at the beginning of the scheduling and after the failure, only the affected task keeps its parameters modified. Moreover, in the FT-PF approach, the additional time for a task recovery is created by increasing the WCET while keeping implicit deadlines, whereas in our approach, the WCET is kept at its initial value and deadlines are constrained.

There are still some works on permanent failures in the context of partitioned scheduling. We can cite [14] whose idea is to provide each core with a twin and assign to that twin all the task assigned to the initial core. Then each pair of cores can suffer a failure without any deadlines being missed. However, such a pairing approach can require more cores than necessary and leads to system overload. [15] enriches PB technique by distinguishing two types of backup copy: active backup and passive backup. The active backup is released when we know that there is not enough remaining time to complete the primary copy. So, it can start running even before we know that the primary has failed. The passive backup starts after the failure of the primary. In summary, the active backup is used as a preventive solution and the passive as a curative.

3.2 Originality of our contribution

PB which is the main technique used to manage a permanent failure, is not suitable in a global scheduling context, because it assumes that primary copies and backups are assigned statically to the cores and no task migration is allowed. Due to backup scheduling, PB technique causes an increase of system load and therefore of the number of redundant cores required to guarantee the feasibility. In light of the previous remarks, building a tolerant scheduling with PFair requires a different approach. We thus propose a technique based on the modification of the temporal parameters. Since tasks are divided into subtasks, no need to use ckeck pointing for the recovery. The originality of this contribution is thus at two levels: 1- It covers a domain where there is little work: the domain of Failure tolerance to a permanent failure in a context of global scheduling, specifically by using Pfair algorithms; 2- It offers a new approach that limits hardware redundancy while preserving the full functionality of the system.

4 Our approach

To guarantee a valid behavior despite the failure of a core, we propose to act on the one hand at the architecture level, through the limited hardware redundancy and, on the other hand, at the system level through dynamic reconfiguration of feasibility windows. The main problem of the latter is that it can produce priority modification (inversion) that must be managed.

4.1 Limited hardware redundancy

The minimal required number of cores for the application to be feasible is $m = \lceil U \rceil$ according to the feasibility condition. Now, if $U = m$, the system cannot support an additional execution time unit on m cores, since S is fully loaded. Thus, if U is an integer, we set $m = U + 1$. Therefore, we finally state $m = \lfloor U \rfloor + 1$. The limited hardware redundancy consists in providing one more core than necessary. So, S will run on $m + 1$ cores. When a failure occurs, after recovery, the system will remain feasible on the m remaining cores.
We denote t_p the time by which the failure occurs. The system thus switch to m cores at time $t_p + 1$.

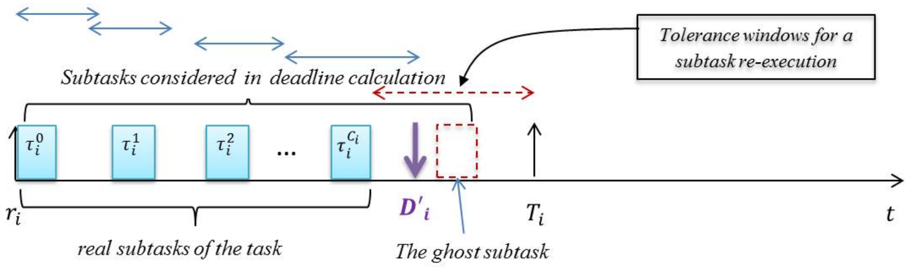
4.2 Dynamic reconfiguration of windows: the ghost subtask method

The objective is to create for each task a feasibility window, called tolerance window, which will be used as additional time to permit the affected subtask rescheduling. For each task, we replace the implicit deadlines with constrained deadlines. To compute them, we use the *ghost subtask method* which simulates the addition of a subtask to each task.

For each task τ_i, we consider a WCET equal to C_i+1 in the calculation of the subtask feasibility windows: each instance is assumed to have one more subtask. The pseudo-deadline of penultimate subtask ($\tau_i^{C_i-1}$) of the first instance is taken as relative deadline: $D_i' = d_i^{C_i-1}$ (see Fig. 1).

The subtask feasibility windows of the first instance of a task τ_i' with constrained deadlines are computed by [16] $r_i'^j = \lfloor \frac{j}{CH_i} \rfloor$ and $d_i'^j = \lceil \frac{j+1}{CH_i} \rceil, j \geq 0$, where $CH_i = \frac{C_i}{D_i'}$ denotes the load factor of the task. To obtain the corresponding parameters for the $h^{th} (h > 0)$ instance of τ_i', we just add $h * T_i$ to the first instance values.

Before the failure, the feasibility windows are calculated with constrained deadlines. When the failure is detected, the subtasks of the unaffected tasks switch to their windows with implicit deadlines, whereas the affected task τ_{i_0} keeps on using the constrained windows until the end of the current hyperperiod. In addition, the WCET of τ_{i_0} switches from C_{i_0} to $C_{i_0} + 1$ to integrate



Tolerance windows for a subtask re-execution
Subtasks considered in deadline calculation
τ_i^0
τ_i^1
τ_i^2
...
$\tau_i^{C_i}$
r_i
D'_i
T_i
t
real subtasks of the task
The ghost subtask

Fig. 1. The Ghost subtask method

the tolerance windows needed for the affected substask re-execution.
We thus use the following systems:

- The *initial system* $S : \tau_i < C_i, T_i >$, having tasks with implicit deadlines;
- The *ghost system* $S^+ : \tau_i < C_i + 1, T_i >$ in which each task has one more substask (the ghost subtask used to determine the task deadline);
- The *constrained system* $S' : \tau'_i < C_i, D'_i, T_i >$ with constrained deadlines deduced from S^+ feasibility windows;
- The *intermediate system* $S_{i0} : \tau_{i \neq i_0} < C_i, T_i >, \tau_{i_0} < C_{i_0} + 1, T_{i_0} >$.

Our assumptions on these systems are the following:

1. S is feasible, which is guaranteed by the choice of m cores.
2. Each task in S has at least one left time unit between its WCET and its period. i.e $\forall \tau_i, T_i - C_i \geq 1$.
3. S' is feasible on $m+1$ cores: $\sum_{i=1}^{n}(C_i/D'_i) < m+1$ [16].
4. S_{i_0} is feasible on m cores for any i_0. i.e $\max_{i_0}(\sum_{i \neq i_0}(C_i/T_i) + \frac{C_{i_0}+1}{T_{i_0}}) \leq m$.

The subtasks use their constrained feasibility windows before failure, then the intermediate windows from the time of failure detection $t_p + 1$ until the next hyper-period of the system $Next_H$, and finally, their initial windows after this hyper-period.

Formally, reconfiguration is performed as follows:

Non-affected tasks $\tau_i (i \neq i_0)$:
Not yet scheduled subtasks switch to their windows in S: $[r'^j_i, d'^j_i) \longmapsto [r^j_i, d^j_i)$.

Affected task τ_{i_0}: three steps.
Before H, the remaining subtasks keep their constrained feasibility windows:
$[r'^j_{i_0}, d'^j_{i_0}) \longmapsto [r'^j_{i_0}, d'^j_{i_0})$;
After H, the remaining subtasks switch to their windows in S:
$[r'^j_{i_0}, d'^j_{i_0}) \longmapsto [r^j_{i_0}, d^j_{i_0})$;
The affected subtask $\tau^{j_0}_{i_0}$ is rescheduled in the tolerance windows of the current instance: $[r'^{j_0}_{i_0}, d'^{j_0}_{i_0}) \longmapsto [r^{t_h}_{i_0}, d^{t_h}_{i_0})$.

In the sequel, we denote by $S' \longrightarrow_{t_p} S_{i_0}$ the system running from time 0 to $Next_H$, with failure at time t_p.

4.3 Justification of the approach

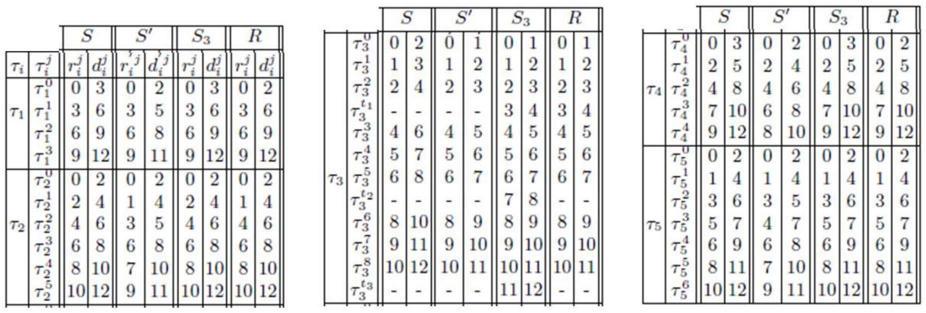
Consider first the architecture. For the sake of weight, cost and energy consumption, we decided to use the lowest possible number of cores. Thus, we only add one core. Then, we could have decided to use it as a spare core: the application runs on m cores and when one of them fails, the spare one takes over. But since we wanted to be able to give additionnal time to the affected task, we prefered to fully use the $(m+1)$ cores to get free slots for the additional processing time unit.

At the system level, we wanted to constrain the system as less as possible, in order to manage the largest number of feasible systems. Therefore, we considered the three different systems. The system S' with constrained deadlines but with the initial WCET. Starting with the increased WCET system S^+ would have increased the load factor. Thus, some actually feasible systems would have become non feasible. Then the intermediate system S_{i_0}, whose feasibility windows are constrained only for one task and relaxed for the other ones, provides more flexibility to reschedule the affected subtask. Finally, return to the initial system at the hyper-period ensures the feasibilty of the system on the remaining cores after recovering.

4.4 Illustration of the approach

Consider the following system of tasks :
$S : \tau_1 < 1, 3 >, \tau_2 < 3, 6 >, \tau_3 < 3, 4 >, \tau_4 < 5, 12 >, \tau_5 < 7, 12 >$.
We have $U = \frac{32}{12}$ thus $m = 3$ and $h = 12$.

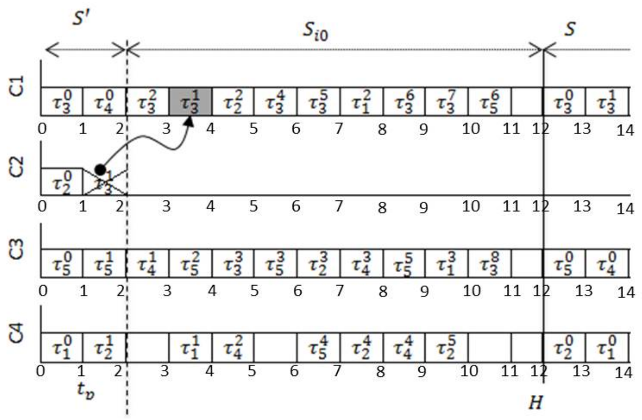
We first add a 4th core. We then compute constrained deadlines. Each task is supposed to have a WCET increased by 1. We consider the first instance of each task and get $(D'_i = d_i^{C_i - 1})$:
$D'_1 = d_1^0 = \lceil \frac{0+1}{\frac{2}{3}} \rceil = 2$; $D'_2 = d_2^2 = \lceil \frac{2+1}{\frac{4}{6}} \rceil = 5$; $D'_3 = d_3^2 = \lceil \frac{2+1}{\frac{4}{4}} \rceil = 3$;
$D'_4 = d_4^4 = \lceil \frac{4+1}{\frac{6}{12}} \rceil = 10$; $D'_5 = d_5^6 = \lceil \frac{6+1}{\frac{8}{12}} \rceil = 11$.
We obtain the following constrained system:
$S' : \tau'_1 < 1, 2, 3 >, \tau'_2 < 3, 5, 6 >, \tau'_3 < 3, 3, 4 >, \tau'_4 < 5, 10, 12 >, \tau'_5 < 7, 11, 12 >$.
We assume that a failure occurs on the core C2 at time 1, thus is detected at time 2. The affected task is τ_3 (i.e $i_0 = 3$). We thefore have
$Si_0 : \tau_1 < 1, 3 >, \tau_2 < 3, 6 >, \tau_{3_0} < 4, 4 >, \tau_4 < 5, 12 >, \tau_5 < 7, 12 >$.
Figure 2 shows the feasibility windows of each subtask in the different systems S, S' and S_3 and the actual feasibility windows used by each substask during the scheduling (see column R).
In S_3, for each instance h of τ_3 an additionnal subtask $\tau_3^{t_h}$ is provided which feasibility windows can be used as tolerance window for re-execution. Thus, the following windows are reserved: $[3, 4)$ in the first instance, $[7, 8)$ in the second and $[11, 12)$ in the third. Since the affected subtask belongs to the first instance of τ_3, the first tolerance window $[3, 4)$ is used for its rescheduling. Figure 3 shows the tolerant scheduling of system S where the failure on the core $C2$ is detected at time $t_p + 1 = 2$. An additional subtask is scheduled in the tolerance windows $[3, 4)$



		S		S'		S_3		R	
τ_i	τ_i^j	r_i^j	d_i^j	$r_i^{'j}$	$d_i^{'j}$	r_i^j	d_i^j	r_i^j	d_i^j
τ_1	τ_1^0	0	3	0	2	0	3	0	2
	τ_1^1	3	6	3	5	3	6	3	6
	τ_1^2	6	9	6	8	6	9	6	9
	τ_1^3	9	12	9	11	9	12	9	12
τ_2	τ_2^0	0	2	0	2	0	2	0	2
	τ_2^1	2	4	1	4	2	4	1	4
	τ_2^2	4	6	3	5	4	6	4	6
	τ_2^3	6	8	6	8	6	8	6	8
	τ_2^4	8	10	7	10	8	10	8	10
	τ_2^5	10	12	9	11	10	12	10	12
τ_3	τ_3^0	0	2	0	1	0	1	0	1
	τ_3^1	1	3	1	2	1	2	1	2
	τ_3^2	2	4	2	3	2	3	2	3
	$\tau_3^{t_1}$	-	-	-	-	3	4	3	4
	τ_3^3	4	6	4	5	4	5	4	5
	τ_3^4	5	7	5	6	5	6	5	6
	τ_3^5	6	8	6	7	6	7	6	7
	$\tau_3^{t_2}$	-	-	-	-	7	8	-	-
	τ_3^6	8	10	8	9	8	9	8	9
	τ_3^7	9	11	9	10	9	10	9	10
	τ_3^8	10	12	10	11	10	11	10	11
	$\tau_3^{t_3}$	-	-	-	-	11	12	-	-
τ_4	τ_4^0	0	3	0	2	0	3	0	2
	τ_4^1	2	5	2	4	2	5	2	5
	τ_4^2	4	8	4	6	4	8	4	8
	τ_4^3	7	10	6	8	7	10	7	10
	τ_4^4	9	12	8	10	9	12	9	12
τ_5	τ_5^0	0	2	0	2	0	2	0	2
	τ_5^1	1	4	1	4	1	4	1	4
	τ_5^2	3	6	3	5	3	6	3	6
	τ_5^3	5	7	4	7	5	7	5	7
	τ_5^4	6	9	6	8	6	9	6	9
	τ_5^5	8	11	7	10	8	11	8	11
	τ_5^6	10	12	9	11	10	12	10	12

Fig. 2. Subtask feasibility windows in different systems

to fully complete τ_3. From time 2, the other tasks use their relaxed feasibility windows.



S'
S_{i0}
S
C1
C2
C3
C4
τ₃⁰ τ₄⁰ τ₃² τ₃¹ τ₂² τ₃⁴ τ₃⁵ τ₁² τ₃⁶ τ₃⁷ τ₅⁶ τ₃⁰ τ₃¹
τ₂⁰ τ₃¹
τ₅⁰ τ₅¹ τ₄¹ τ₅² τ₃³ τ₅³ τ₂³ τ₄³ τ₅⁵ τ₁³ τ₃⁸ τ₅⁰ τ₄⁰
τ₁⁰ τ₂¹ τ₁¹ τ₄² τ₅⁴ τ₂⁴ τ₄⁴ τ₂⁵ τ₂⁰ τ₁⁰
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
t_v
H

Fig. 3. A failure-tolerant scheduling of System S

5 Some issue: inversion of priorities

5.1 The problem

According to PD2, priorities among subtasks are based on their feasibility windows. Because these windows are not the same in systems S, $S'$and S_{i_0}, it may result in our approach changes of priorities between subtasks while switching from one system to another. Moreover, a subtask may be released later in S_{i_0} than in S'. This priority inversion between subtasks has consequences on the scheduling after the failure. Even if after reconfiguration subtasks have switched

to their windows in the system S_{i_0}, $S' \longrightarrow_{t_p} S_{i_0}$ doesn't necessarily behave identically to S_{i_0} for three main reasons: firstly, at time $t > t_p$ there may exist subtasks pending in S_{i_0}, which have already been scheduled in S'. We call them *anticipated subtasks*. These subtasks are pending in S_{i_0} but not in $S' \longrightarrow_{t_p} S_{i_0}$. For example, τ_5^1 and τ_2^1 in Fig 3 and τ_0^0, τ_1^0 and τ_2^0 in Fig 5. Secondly, there may exist subtasks scheduled in S_{i_0} before the failure that were planned after the failure in S'. We name them the *staggered subtasks*. These are pending in $S' \longrightarrow_{t_p} S_{i_0}$, but not in S_{i_0}. It is the case of τ_8^0 and τ_9^0 in Fig 5. Finally, a subtask can be scheduled at any time t in S_{i_0} but later in $S' \longrightarrow_{t_p} S_{i_0}$ because it was replaced by a higher priority subtask. We call it a *postponed subtask*. In Fig 5 τ_{17}^0 and τ_{18}^0 are postponed subtasks.
Note that, a subtask is said anticipated, staggered or postponed by reference to the time planned for its execution in S_{i_0}.

5.2 Illustration

Consider the behavior of the systems $S' \longrightarrow_1 S_{i_0}$ (Fig. 3) and S_{i_0} (Fig. 4). Due to reconfiguration, substasks have the same feasibility windows in both systems. Moreover, in this example, there is no staggered substask, since at time $t = 1$ all substasks already scheduled in S_{i_0} have already been scheduled in $S' \longrightarrow_1 S_{i_0}$.

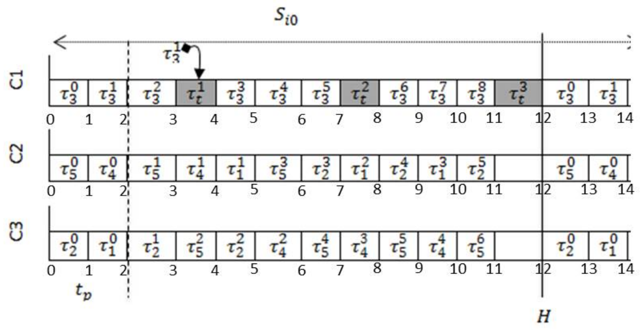


Fig. 4. S_{i_0} scheduling on a 3 cores

We have following first remark. *If there is no staggered subtask, a subtask τ_i^j is scheduled in $S' \longrightarrow_{t_p} S_{i_0}$ at the same time than in S_{i_0} (eg. τ_3^2 and τ_5^2) or earlier (eg. τ_4^1 and τ_1^1) .*

Let's now study an example with staggered subtask. Consider the following system composed of 48 tasks. $\tau_{i-j} < C, T >$ denotes a list of tasks $\tau_i, \tau_{i+1}...\tau_j$ with common temporal parameters.
$S1 : \tau_{0-3} < 1, 20 >, \tau_{4-7} < 1, 36 >, \tau_{8-47} < 2, 38 >$.
$U = 2.41$ and $m = \lfloor 2.41 \rfloor + 1 = 3$. Then, the system is feasible on 3 cores.

Applying the ghost subtask method we get the constrained system
$S1' = \tau_{0-3} < 1, 10 >, \tau_{4-7} < 1, 18 >, \tau_{8-47} < 2, 26 >$ which runs on 4 cores.

We suppose that a failure occurs at time $t_p = 0$ on core C4 affecting the subtask of τ_3^0. The feasibility windows of the first instances of the tasks in different systems are given below:
$S1$: $\tau_{0-3}^0(0, 20), \tau_{4-7}^0(0, 36), \tau_{8-47}^0(0, 19), \tau_{8-47}^1(19, 38)$.
$S1'$: $\tau_{0-3}^0(0, 10), \tau_{4-7}^0(0, 18), \tau_{8-47}^0(0, 13), \tau_{8-47}^1(13, 26)$.
$S1_3$: $\tau_{0-2}^0(0, 20), \tau_3^0(0, 10), \tau_3^1(10, 20), \tau_{4-7}^0(0, 36), \tau_{8-47}^0(0, 19), \tau_{8-47}^1(19, 38)$.
We can note the priority inversion between τ_{0-3}^0 and τ_{8-47}^0: in $S1$ and $S1_3$, τ_{8-47}^0 has priority over τ_{0-3}^0, whereas τ_{0-3}^0 has priority over τ_{8-47}^0 in $S1'$. Figure 5 shows respectively the PD2 schedule for the systems $S1' \longrightarrow_0 S1_3$ and $S1_3$.

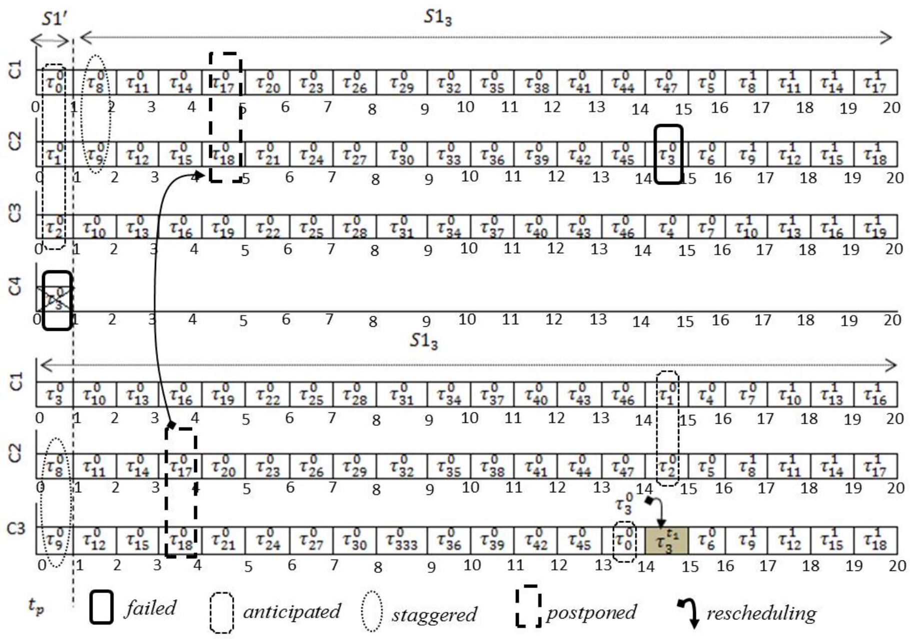


Fig. 5. A Tolerant Scheduling with staggered subtasks

The comparison of two figures leads us to a second remark:
at time t_p, the 3 anticipated substasks τ_0^0, τ_1^0 and τ_2^0 correspond to the 2 staggered subtasks τ_8^0 and τ_9^0. In $S1' \longrightarrow_0 S1_{i_0}$, the staggered subtasks are scheduled at time $t_p + 1 = 1$ causing the postponement of the substasks (τ_{14}^0 and τ_{15}^0) scheduled at this time in $S1_{i_0}$. The 2 postponed subtasks are scheduled at time 2 and 2 other substasks (τ_{17}^0 and τ_{18}^0) are at their turn potsponed. Subtask postponement ends at the time $t = 15$. From this time, subtasks are scheduled in $S1' \longrightarrow_0 S1_{i_0}$ at the same time as in $S1_{i_0}$ or earlier.

Theses remarks give the key arguments for the proof of the validity of our approach, specifically for Proposition 1 (for the first remark) and for Proposition 2 (for the second remark).

6 The results

6.1 Experimentations

To experiment our approach, we designed a software prototype, named FTA (Failure Tolerance Analyser), to simulate PD2 scheduling with failure. This prototype has three main modules: a random generator systems, a scheduler (that randomly determines the time of the failure and the affected core and schedules the system according to the proposed approach) and a Diagnostic tool (that analyzes the sequence produced by the scheduler and determines whether it is valid and fair).

Several parameters can be set before the analysis: the system load, the number of heavy tasks (i.e tasks with $U_i \geq 0.5$), the time of occurrence of the failure and the affected core. 550 random systems have been generated and submitted to the simulation. This has been done by means of 11 random generations each composed of 50 systems containing different numbers of heavy tasks. The simulation is repeated many times to let the parameters vary and observe their impact. All the obtained scheduling results were valid.

In the following paragraph, we give some elements of proof of this result.

6.2 Validity of our approach

We first define some terms and used notations.

- $Sched_{nb}^{Sys}$: PD2 schedule of the system Sys on nb cores.
- $Sched_{(m+1)\longrightarrow m}^{S'\longrightarrow_{t_p} S_{i_0}}$: PD2 schedule of system S' running on $m+1$ cores with a failure of one core at time t_p and a switch to system S_{i_0} at time t_p+1.
- $Pending(S,t)$: list of pending substasks in system S at time t. In our context, a subtask τ_i^j is pending at time t if it is released and not yet scheduled. And if $j>0$, the previous subtask τ_i^{j-1} has already been scheduled.
- $Exec(\tau_i^j, Sched)$: execution time of the subtask τ_i^j in the schedule $Sched$.
- t_p: failure time; $\tau_{i_0}^{j_0}$: the affected substask.
- $Next_H$: next hyperperiod after t_p;
- $r_i^{j(S)}$ and $d_i^{j(S)}$: pseudo-release date and pseudo-deadline of τ_i^j in system S.

We assume that $Sched_m^S$, $Sched_{(m+1)}^{S'}$ and $Sched_m^{S_{i_0}}$ are valid and fair.

Our main result is that if S, S' and S_{i_0} are feasible respectively on m, $m+1$ and m cores, then $S' \longrightarrow_{t_p} S_{i_0}$ is feasible.
The proof of the validity of $S' \longrightarrow_{t_p} S_{i_0}$ is very technical; thus, we only present some elements in two cases, as stated in the following theorem:

Theorem 1. *If there is no staggered subtask or if there are staggered subtasks and the failure occurs at a time of a hyper-period (i.e $t_p = 0[H]$),*
then $Sched_{(m+1)\longrightarrow m}^{S'\longrightarrow_{t_p} S_{i_0}}$ *is valid and fair.*

To prove this theorem, we first consider the case where no priority inversion takes place, i.e there is no staggered subtasks. This can be expressed by the following condition [H1]:

$$[H1] : \forall \tau_i^j, Exec(\tau_i^j, Sched_m^{S_{i_0}}) \leq t_p \Longrightarrow Exec(\tau_i^j, Sched_{(m+1)}^{S'}) \leq t_p, \tau_i^j \neq \tau_{i_0}^{t_h}.$$

We then state the following proposition:

Proposition 1. *R(t)*
If there is no staggered subtask, at any time $t > t_p$, a subtask τ_i^j is not scheduled later in $Sched_{(m+1)\longrightarrow m}^{S'\longrightarrow_{t_p} S_{i_0}}$ *than in* $Sched_m^{S_{i_0}}$.

To prove this proposition, we proceed by induction on t, using two further properties:

Property 1. Prop1(t): A subtask pending at time t ($t \geq t_p + 1$) in S_{i_0} is either also pending in $S' \longrightarrow_{t_p} S_{i_0}$ or has already been processed.
$\tau_i^j \in Pending(S_{i_0}, t) \Longrightarrow \tau_i^j \in Pending(S' \longrightarrow_{t_p} S_{i_0}, t)$ or
$Exec(\tau_i^j, Sched_{(m+1)\longrightarrow m}^{(S'\longrightarrow_{t_p} S_{i_0})}) < t$

Property 2. Prop2(t)
At time t ($t \geq t_p + 1$), if there are k pending substasks with higher priority than τ_i^j in $S' \longrightarrow_{t_p} S_{i_0}$, then in S_{i_0} there are at least k pending substasks with higher priority than τ_i^j.

Then, we consider the case where there are some priority inversions i.e, some staggered subtasks. For simplicity reason, we present the case where the failure occurs at the beginning of an hyper-period (i.e $t_p = 0[H]$). In this case, the first subtask of each task is pending in S'. We then prove the next proposition:

Proposition 2. *Prop(t)*
At any time $t > 0$, if all the subtasks already scheduled in S_{i_0} have already been scheduled in $S' \longrightarrow_{t_p} S_{i_0}$, then from this time, each subtask is scheduled in $S' \longrightarrow_{t_p} S_{i_0}$ not later than as in S_{i_0}. i.e
If $Exec(\tau_i^j, Sched_m^{S_{i_0}}) = t'$ *with* $t' > t_p$ *then,* $Exec(\tau_i^j, Sched_{(m+1)\longrightarrow m}^{(S'\longrightarrow_{t_p} S_{i_0})}) \leq t'$.

With these two propositions we now prove the theorem.

Proof. (Theorem 1)
- First case: there is no staggered subtask.
According to $R(t)$, after the failure we have
$Exec(\tau_i^j, Sched_{(m+1)\longrightarrow m}^{(S'\longrightarrow_{t_p} S_{i_0})}) \leq Exec(\tau_i^j, Sched_m^{S_{i_0}})$.
Since $Sched_m^{S_{i_0}}$ is valid and fair by assumption, Theorem 1 holds.

- Second case: there are k staggered subtasks at $t_p = 0$.
Using the second remark of Paragraph 5.2 and $Prop(t)$, we prove that:

(1) At time $t_p = 0$, to the k staggered substasks $\tau^0_{a_1}...\tau^0_{a_k}$ correspond $k+1$ anticipated substasks $\tau^0_{b_1}...\tau^0_{b_{k+1}}$ (in decreasing priority order) such that $t_{b_i} = Exec(\tau^0_{b_i}, Sched^{S_{i_0}}_m) > t_p$ and $Exec(\tau^0_{b_i}, Sched^{S'}_{(m+1)}) = t_p$, with $t_{b_1} \leq t_{b_2} \leq ... \leq t_{b_{k+1}}$ and $t_{max} = MAX(t_{b_1})$. (2) At time $t = 1$, the staggered subtasks $\tau^0_{a_1}...\tau^0_{a_k}$ are among the m highest priority subtasks in $S' \longrightarrow_{t_p} S_{i_0}$ and are thus executed while meeting their deadlines: in fact, $Exec(\tau^0_{a_i}, Sched^{S'}_{(m+1)}) > 0 \Longrightarrow d^{0(S')}_{a_i} > 1$ and thus $d^{0(S)}_{a_i} > 1$. Their execution can lead to the postponment of other planned subtasks $\tau^{j_{ir}}_{u_{ir}}$. (3) From time t=2 to time t= $t_{max}-1$, substasks postponed at $t-1$ are scheduled at t. We thus have a cascade of postponements of subtasks which are shifted by one time unit. And we prove that they still meet their pseudo-deadlines, because of the assumption $U(S_{i_0}) \leq m$ (for reasons of space, we cannot detail the proof here). (4) At time t_{max}, all the staggered and postponed subtasks have been scheduled in place of anticipated subtasks. Now, all the substasks already scheduled in S_{i_0} are already scheduled in $S' \longrightarrow_{t_p} S_{i_0}$ and thus $Prop(t_{max})$ can be applied.

Since $Prop(t)$ is true and $Sched^{S_{i_0}}_m$ is valid by assumption $[A3]$, all the pseudo-deadlines are met in $S' \longrightarrow_{t_p} S_{i_0}$. Thus, $Sched^{(S' \longrightarrow_{t_p} S_{i_0})}_{(m+1) \longrightarrow m} < d^{j(S)}_i$ is valid and Theorem 1 holds ■

7 Conclusion and perspectives

Failure tolerance is a fundamental aspect of embedded system design. Today, many such systems run on multicore architectures. The advent of multicore brings timeliness due to increased processing units. However, there are some issues linked to these architectures, such as hardware failure risks. In this paper, we considered a permanent failure of one core detected within the next time unit after its occurrence for systems scheduled under PD2. Then one subtask is affected and its execution is resumed. We proposed an approach for failure tolerance based on limited hardware redundancy and dynamic reconfiguration of subtask feasibility windows. We have experimentaly tested this method and the resulting schedules were always valid and fair whatever the system load, the time of the failure and the percentage of heavy tasks. This approach raises the problem of priority inversion between subtasks during the scheduling. When there is no staggered subtask, we proved the validity of the resulting schedule. We also proved it when there are some staggered subtasks and the failure occurs at the time of a hyper-period (i.e $t_p = 0[H]$). We are currently completing the proof for the last case where there are staggered subtasks and the failure occurs at any time different of a hyperperiod. This part of the proof is more technical than the one presented in this paper.
In future works we will study the case where the failure detection delay is larger, and therefore more subtasks, that may belong to different tasks, are re-executed. We will also study the case where several cores may fail.

References

1. C. Lee, H. Kim, H. Park, S. Kim, H. Oh, S. Ha : A Task Remapping Technique for Reliable Multicore Embedded Systems. International Conference on Hardware/Software Codesign and System Synthesis, pages 307 to 316 (2015)
2. S. Malhotra, P. Narkhede, K. Shah, S. Makaraju, M. Shanmugasundaram: A Review of Fault Tolerant Sscheduling in Multicore Systems. International Journal of Scientific and Technology Research, Volume 4, pages 132 to 136 (2015)
3. Y. Mouafo, A. Choquet-Geniet, G. Largeteau-Skapi: Failure Tolerance for a Multicore Real-Time System Scheduled by PD2. Proceedings of the 9_{th}, Junior Researcher Workshop on Real-Time Computing, pages 1-4 (2015).
4. J. Anderson, A. Srinivavasan: A New Look at Pfair Priorities. Rap. tech. TR00-023, University of North Carolina at Chapel Hil (1999)
5. S.K. Baruah, N.K. Cohen, C.G. Plaxton, D.A. Varvel: A Notion of Fairness in Ressource Allocation. Algorithmica 15 (6), Pg 600-625 (1996)
6. S. Shiravi, M.E. Salehi: Fault Tolerant Task Scheduling Algorithm for Multicore Systems. The 22_{nd} Iranian Conference on Electrical Engineering (ICEE), Pg 885-890 (2014)
7. E. Chantery, Y. Pencole: Modélisation et Intégration du Diagnostic Actif dans une Architecture Embarqué. In: Journal Européen des Systèmes Automatisées, MSR 2009, Pg 789-803 (2009)
8. A. Bondavalli, S. Chiaradonna, F. Di Giandomenico: Efficient Fault Tolerance: An Approach to Deal With Transient Faults in Multiprocessor Architectures. International Conference on Parallel and Distributed Systems, Pg 354-359 (1994)
9. M. Cirinei, E. Bini, G. Lipari, A. Ferrari: A Flexible Scheme for Scheduling Fault-Tolerant Real-Time Tasks on Multiprocessors. IEEE International Parallel and Distributed Processing Symposium, Pg 1-8 (2007)
10. C.M. Krishna: Fault-Tolerant Scheduling In Homogeneous Real-Time Systems. In: Journal ACM Computing Surveys (CSUR), Volume 46 Issue 4, Article No 48 (2014)
11. K. Ahn, J. Kim, S. HONG: Fault-Tolerant Real-Time Scheduling Using Passive Replicas. Pacific Rim International Symposium on Fault-Tolerance, Pg 98-103 (1997)
12. A. Christy, T.R. Gopalakrishnan Nair: Fault-Tolerant Real Time Systems: International Conference on Managing Next Generation Software Application (2008)
13. F. Liberato, S. Lauzac, R. Melhem, D. Mosse: Fault-Tolerant Real-Time Global Scheduling on Multiprocessors. In: Proceedings of the 11th Euromicro Conference on Real-time Systems, Pg 252-259 (1999)
14. Y. Oh, S.H. Son: An Algorithm For Real-Time Fault-Tolerant Scheduling in Multiprocessor Systems. Euromicro Workshop on Real-Time Systems, Pages 190-195 (1992)
15. A.A. Berossi, L.V. Mancini, F. Rossini: Fault-Tolerant Rate-Monotonic First-Fit Scheduling in Hard Real-Time Systems. IEEE Transactions on Parallel and Distributed Systems 10, Pg 934-945 (1999)
16. S. Malo, A. Choquet-Geniet, M. Bikienga: PFair Scheduling of Late Released Tasks with Constrained Deadlines. Colloque National sur la Recherche en Informatique et ses applications, pages 142-149 (2012)