Synthesis of Flexible Fault-Tolerant Schedules with Preemption for Mixed Soft and Hard Real-Time Systems

Viacheslav Izosimov [1], Paul Pop [2], Petru Eles [1], Zebo Peng [1]

[1] *Dept. of Computer and Information Science*
Linköping University
SE-581 83 Linköping, Sweden
E-mail: {viaiz | petel | zebpe}@ida.liu.se

[2] *Dept. of Informatics and Mathematical Modelling*
Technical University of Denmark
DK-2800 Kongens Lyngby, Denmark
E-mail: Paul.Pop@imm.dtu.dk

Abstract

In this paper we present an approach for scheduling with preemption for fault-tolerant embedded systems composed of soft and hard real-time processes. We are interested to maximize the overall utility for average, most likely to happen, scenarios and to guarantee the deadlines for the hard processes in the worst case scenarios. In many applications, the worst-case execution times of processes can be much longer than their average execution times. Thus, designs for the worst-case can be overly pessimistic, i.e., result in low overall utility. We propose preemption of process executions as a method to generate flexible schedules that maximize the overall utility for the average case while guarantee timing constraints in the worst case. Our scheduling algorithms determine off-line when to preempt and when to resurrect processes. The experimental results show the superiority of our new scheduling approach compared to approaches without preemption.

1. Introduction

Fault-tolerant embedded real-time systems have to meet their deadlines and function correctly in the worst-case and under presence of faults. Such systems are usually designed for the worst-case, which often leads to overly pessimistic solutions since the worst-case execution times of processes can be much longer than their average execution times [1]. Design of fault-tolerant embedded real-time systems for the average case, addressed in this paper, is a promising alternative to the purely worst-case-driven design. It is important to emphasize that the generated designs have to be safe, i.e. all deadlines are met, even in the worst-case execution scenarios and when affected by faults.

Faults can be permanent (i.e. damaged microcontrollers or communication links), transient, or intermittent. Transient and intermittent faults (also known as "soft errors") appear for a short time and can be caused by electromagnetic interference, radiation, temperature variations, software "bugs", etc. Transient and intermittent faults[1], which we will deal with in this paper, are the most common and their number is increasing due to greater complexity, higher frequency and smaller transistor sizes [10].

1. We will refer to both transient and intermittent faults as "transient" faults.

Real-time systems have been classified as *hard* real-time and *soft* real-time systems. For hard real-time processes, failing to meet a deadline can potentially have catastrophic consequences, whereas a soft real-time process retains some diminishing value after its deadline. Traditionally, hard and soft real-time systems have been scheduled using very different techniques [12]. However, many applications have both hard and soft timing constraints [3], and therefore researchers have proposed techniques for addressing mixed hard/soft real-time systems [3, 5, 4].

In the context of hard real-time systems, researchers have shown that schedulability can be guaranteed for online scheduling [7, 13, 18]. However, such approaches lack the predictability required in many safety-critical applications, where static off-line scheduling is the preferred option for ensuring both the predictability of worst-case behavior, and high resource utilization. Thus, researchers have proposed approaches for integrating fault tolerance into the framework of static scheduling. A heuristic for combining together several static schedules in order to mask fault patterns through replication is proposed in [15]. The actual static schedules are generated according to the approach in [6]. Xie et al. [17] propose a technique to decide how replicas can be selectively inserted into the application, based on process criticality. Kandasamy et al. [11] propose constructive mapping and scheduling algorithms for transparent re-execution on multiprocessor systems. In [8] we have shown how re-execution and active replication can be combined in an optimized implementation that leads to a schedulable fault-tolerant application without increasing the amount of employed resources.

Regarding soft real-time systems, researchers have shown how faults can be tolerated with active replication while maximizing the utility of the system [14]. In [2] faults are tolerated while maximizing the reward in the context of online scheduling and an imprecise computation model, where processes are composed of mandatory and optional parts. In [16] trade-off between performance and fault-tolerance, based on active replication, is considered in the context of online scheduling. This, however, incurs a large overhead during runtime which seriously affects the quality of results.

In [9], we have considered embedded systems composed of both hard and soft processes. Process re-execution is used to provide the required level of fault tolerance. We have proposed a novel quasi-static scheduling strategy, where a set of fault-tolerant schedules is synthesized off-line and, at run time, the scheduler will select the right schedule based on the occurrence of faults and the actual execution times of processes, such that hard deadlines are guaranteed and the overall system utility is maximized. The online overhead of quasi-static scheduling is very low, compared to traditional online scheduling approaches [4]. The proposed scheduling strategy can also handle overload situations with dropping of soft processes. Dropping allows to skip execution of a soft process if such an execution leads to violation of hard deadlines or to deterioration of the produced overall utility. The dropping technique, however, only provides two extreme alternatives: complete execution of a soft process or skipping of its execution. This can result in a very pessimistic schedule, especially, if the worst-case execution times of processes are much longer than their average case execution times.

In this paper, we enhance our fault tolerance scheduling strategy with preemption, in order to generate flexible schedules that allow to preempt execution of a process and then resurrect the process if that is needed and profitable. Flexible schedules with preemption overcome the pessimism of previous approaches while generating safe schedules even in the worst-case overloading situations and under presence of faults. We propose static and quasi-static algorithms that generate off-line schedule tables, which then, at run time, are used to safely preempt and resurrect processes when executing the application.

The next section presents our application model, time/utility model, and the fault tolerance techniques. In Section 3, we compare scheduling without and with preemption. Section 4 outlines our problem formulation and Section 5 presents our heuristics for static and quasi-static scheduling. Experimental results, which demonstrate advantages of flexible schedules with preemption, are presented in Section 6.

2. Application Model

We model an application $\mathcal{A}$ as a set of directed, acyclic, polar graphs $\mathcal{G}_k(\mathcal{V}_k, \mathcal{E}_k) \in \mathcal{A}$. Each node $P_i \in \mathcal{V}_k$ represents one process. An edge $e_{ij} \in \mathcal{E}_k$ from P_i to P_j indicates that the output of P_i is the input of P_j. A process can be activated after all its inputs, required for the execution, have arrived. The process issues its outputs when it terminates. Processes can be preempted during their execution.

We consider that the application is running on a single computation node. Each process P_i in the application has a best-case execution time (BCET), t_i^b, and a worst-case execution time (WCET), t_i^w. The execution time distribution $E_i(t)$ of process P_i is given. An average-case execution time (AET) for process P_i, t_i^e, is obtained from the execution time distribution $E_i(t)$. The communication time between processes is considered to be part of the process execution time and is not modeled explicitly. In Fig. 1 we have an application $\mathcal{A}$ consisting of the process graph $\mathcal{G}_1$ with three processes, P_1, P_2 and P_3. The execution times for the processes are shown in the table. μ is a recovery overhead, which represents the time needed to start re-execution of a process in case of faults. ζ is a preemption overhead, which represents the time needed to preempt a process and store its state (to "checkpoint"). ρ is a resurrecting overhead, which represents the time needed to continue execution of a preempted process, including restoring the process state.

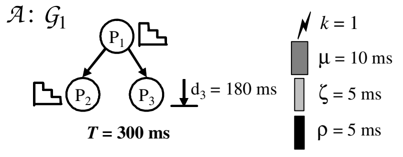
All processes belonging to a process graph $\mathcal{G}$ have the same period $T = T_{\mathcal{G}}$, which is the period of the process graph. In Fig. 1 process graph $\mathcal{G}_1$ has a period $T = 300$ ms. If process graphs have different periods, they are combined into a hyper-graph capturing all process activations for the hyper-period (LCM of all periods).

2.1 Utility Model

The processes of the application are either hard or soft. We will denote with $\mathcal{H}$ the set of hard processes and with $\mathcal{S}$ the set of soft processes. In Fig. 1 processes P_1 and P_2 are soft, while process P_3 is hard. Each hard process $P_i \in \mathcal{H}$ is associated with an individual hard deadline d_i. Each soft process $P_i \in \mathcal{S}$ is assigned with a utility function $U_i(t)$, which is any non-increasing monotonic function of the completion time of a process. For example, in Fig. 2a the soft process P_a is assigned with a utility function $U_a(t)$ and completes at 60 ms. Thus, its utility would equal to 20. The overall utility of the application is the sum of individual utilities produced by soft processes. The utility of the application depicted in Fig. 2b, which is composed of two processes, P_b and P_c, is 25, in the case that P_b completes at 50 ms and P_c at 110 ms giving utilities 15 and 10, respectively. Note that hard processes are not associated with utility functions but it has to be guaranteed that, under any circumstances, they meet their deadlines.

Both hard and soft processes can be preempted, as illustrated in Fig. 2c, where the application $\mathcal{A}$ from Fig. 1 is run with preemption. A *hard* process P_i, even if preempted, has to always complete its execution before the deadline and, thus, has to be always resurrected. We will denote with $P_{i\#j}$ the execution of jth part of process P_i. $1 \leq j \leq n_i + 1$, where n_i is the maximum number of preemptions of P_i. Both hard and soft processes can be preempted several times. In Fig. 2c, process P_3 is preempted at 105 ms, and is resurrected at 135 ms.

A *soft* process P_i is not required to be resurrected. For example, process P_1 is preempted at 30 ms and is not resurrected. However, if the execution of soft process P_i was preempted, its utility is 0 unless the process is resurrected, i.e., $U_i(t) = 0$. In Fig. 2c, processes P_1 and P_2 produce utility of 0 after they are preempted at 30 ms and 65 ms. If P_i is resurrected and completed at time t, then its utility is calculated according to its utility function $U_i(t)$. In Fig. 2c, process P_2 is resurrected and finally completes at 135 ms, which gives the utility of 15. Thus, the total utility for this scenario will



	BCET	AET	WCET
P_1	20	50	80
P_2	30	50	70
P_3	40	60	80

Figure 1. Application Example

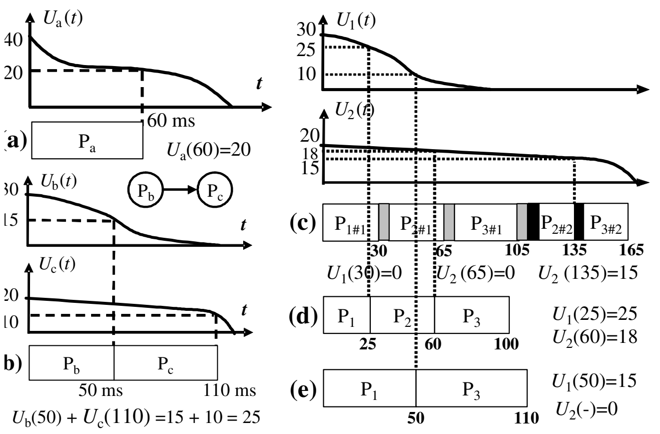


Figure 2. Utility Functions, Preemption, and Dropping

be $U = U_1(30) + U_2(135) = 0 + 15 = 15$. Note that we have accounted for preemption and resurrecting overheads $\zeta = 5$ ms and $\rho = 5$ ms in this application run. If process P_i completes before it is preempted, it will produce utility $U_i(t)$. In the scenario depicted in Fig. 2d, for example, processes P_1 and P_2 complete at 25 and 60 ms, respectively, which gives the utility $U = U_1(25) + U_2(60) = 20 + 18 = 38$.

For a soft process P_i we have the additional option not to start it at all, and we say that we "drop" the process, and thus its utility will be 0, i.e., $U_i(-) = 0$. In the execution in Fig. 2e we drop process P_2 of application $\mathcal{A}$. Thus, process P_1 completes at 50 ms and process P_3 at 110 ms, which gives the total utility $U=U_1(50) + U_2(-) = 10 + 0 = 10$.

Preemption and dropping might be necessary in order to meet deadlines of hard processes, or to increase the overall system utility (e.g. by allowing other, potentially higher-value soft processes to complete).

Moreover, if P_i is preempted or dropped and is supposed to produce an input for another process P_j, we assume that P_j will use an input value from a previous execution cycle, i.e., a "stale" value. This is typically the case in automotive applications, where a control loop executes periodically, and will use values from previous runs if new ones are not available. To capture the degradation of service that might ensue from using stale values, we update our utility model of a process P_i to $U_i^*(t) = \alpha_i \times U_i(t)$, where α_i represents the stale value coefficient. α_i captures the degradation of utility that occurs due to dropping of processes. Thus, if a soft process P_i is preempted (or dropped), then $\alpha_i = 0$, i.e., its utility $U_i^*(t)$ will be 0. If P_i is executed, but reuses stale inputs from one or more of its direct predecessors, the stale value coefficient will be calculated as the sum of the stale value coefficients over the number of P_i's direct predecessors:

$$\alpha_i = \frac{1 + \sum_{P_j \in DP(P_i)} \alpha_j}{1 + |DP(P_i)|}$$

where $DP(P_i)$ is the set of P_i's direct predecessors. Note that we add "1" to the denominator and the dividend to account for P_i itself. The intuition behind this formula is that the impact of a stale value on P_i is in inverse proportion to the number of its inputs.

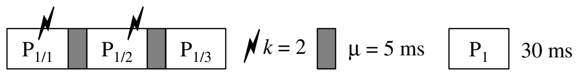


Figure 3. Re-execution

Suppose that soft process P_3 has two predecessors, P_1 and P_2. If P_1 is preempted while P_2 and P_3 are completed successfully, then, according to the formula, $\alpha_3 = (1 + 0 + 1) / (1 + 2) = {}^2/_3$. Hence, $U_3^*(t) = {}^2/_3 \times U_3(t)$. The use of a stale value will propagate though the application. For example, if soft process P_4 is the only successor of P_3 and is completed, then $\alpha_4 = (1 + {}^2/_3) / (1+1) = {}^5/_6$. Hence, $U_4^*(t) = {}^5/_6 \times U_4(t)$.

2.2 Fault Tolerance

In this paper we are interested in techniques for tolerating transient faults, which are the most common faults in today's embedded systems. In our model, we consider that at most k transient faults may occur during one operation cycle of the application.

The error detection and fault-tolerance mechanisms are part of the software architecture. The error detection overhead is considered as part of the process execution time. The software architecture, including the real-time kernel, error detection and fault-tolerance mechanisms are themselves fault-tolerant.

We use re-execution for tolerating faults. Let us consider the example in Fig. 3, where we have process P_1 and $k = 2$ transient faults that can happen during one cycle of operation. In the worst-case fault scenario depicted in Fig. 3, the first fault happens during P_1's first execution, denoted $P_{1/1}$, and is detected by the error detection mechanism. After a worst-case recovery overhead of $\mu = 5$ ms, depicted with a light gray rectangle, P_1 will be executed again. Its second execution $P_{1/2}$ in the worst-case could also experience a fault. Finally, the third execution $P_{1/3}$ of P_1 will take place without fault. In this paper, we will denote with $P_{i/j}$ the jth execution of process P_i in the faulty scenario, where P_i is affected by faults.

Hard processes have to be always re-executed if affected by a fault. Soft processes, if affected by a fault, are not required to recover. A soft process will be re-executed only if it does not impact the deadlines of hard processes, and its re-execution is beneficial for the overall utility.

3. Scheduling with Preemption

In this paper, we will use quasi-static scheduling for generating a tree of schedules with preemption. In [9], we have demonstrated that quasi-static scheduling allows to capture different execution scenarios of an application with soft and hard timing constraints. Each individual schedule in a quasi-static tree is generated with static scheduling.

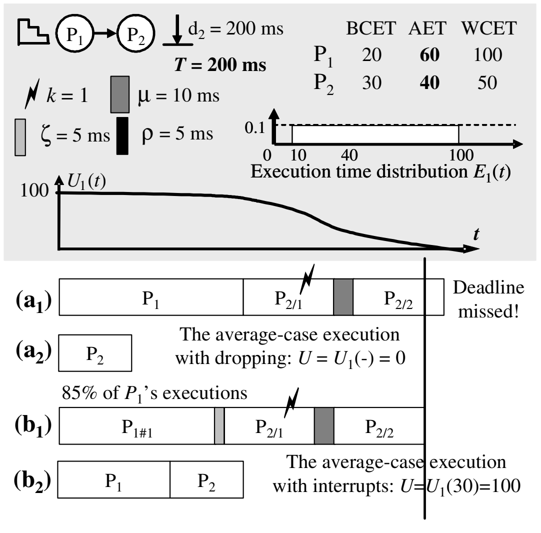
3.1 Preemption vs. Dropping

In this work, we extend our scheduling strategy from [9] with preemption that would allow to stop a process and then resurrect it if needed. In case of dropping we have only two extreme alternatives, complete execution of a soft process or skip its execution (drop process), which can result in a pessimistic schedule, as will be illustrated in the following examples.

In Fig. 4 we show a simple example of two processes P_1 and P_2. P_1 is a soft process with the utility function depicted at the bottom of the grey area and the uniform completion time distribution $E_1(t)$; P_2 is a hard process with a deadline of 200 ms. One fault can happen within a period of 200 ms. If we apply dropping as in Fig. 4a, we have no other option than to drop soft process P_1. This is because in the worst-case, depicted in Fig. 4a$_1$, where processes P_1 and P_2 are executed with their worst-case execution times and process P_2 is re-executed, P_1's execution will lead to missed deadline of P_2. The average case schedule, depicted in Fig. 4a$_2$, will contain only process P_2 and the overall utility of such schedule is 0.

If we apply preemption, as in the worst-case in Fig. 4b, we can preempt process P_1 at time 85 and process P_2 completes its execution before the deadline even if affected by fault. In the average case, the schedule will contain both processes P_1 and P_2 producing the overall utility of 100. Moreover, in 85% of all the cases, process P_1 will complete before we have to preempt it. This would produce a utility of at least 98 in 85% of all the cases. The flexible schedule with preemption clearly outperforms the schedule with only dropping.

Let us consider another example to further illustrate that the safe schedule with dropping and no preemption is pessimistic. In Fig. 5, we consider an application of three processes, P_1 to P_3. P_1 and P_2 are soft processes with utility functions depicted at the bottom of the grey area; P_3 is a hard process with a deadline of 280 ms. The application is run with a period of 330 ms. The execution times are given in the table in the figure. The distribution of execution times $E_1(t)$ and $E_2(t)$ of soft processes P_1 and P_2, respectively, are also depicted. The maximum number of faults is $k = 1$ and the recovery overhead μ is 10 ms for all processes. The safe static schedule with dropping is shown in Fig. 5a. Soft process P_2 is dropped and soft process P_1 is executed completely. This schedule will produce a utility of $U = U_1(30)+U_2(–) = 50 + 0 = 50$ in



d_2 = 200 ms
T = 200 ms
k = 1
μ = 10 ms
ζ = 5 ms
ρ = 5 ms
BCET AET WCET
P_1 20 60 100
P_2 30 40 50
0.1
0 10 40 100
Execution time distribution $E_1(t)$
100 $U_1(t)$
t
P_1 $P_{2/1}$ $P_{2/2}$
Deadline missed!
The average-case execution with dropping: $U = U_1(-) = 0$
85% of P_1's executions
$P_{1\#1}$ $P_{2/1}$ $P_{2/2}$
The average-case execution with interrupts: $U=U_1(30)=100$

Figure 4. Example 1: Preemption vs. Dropping

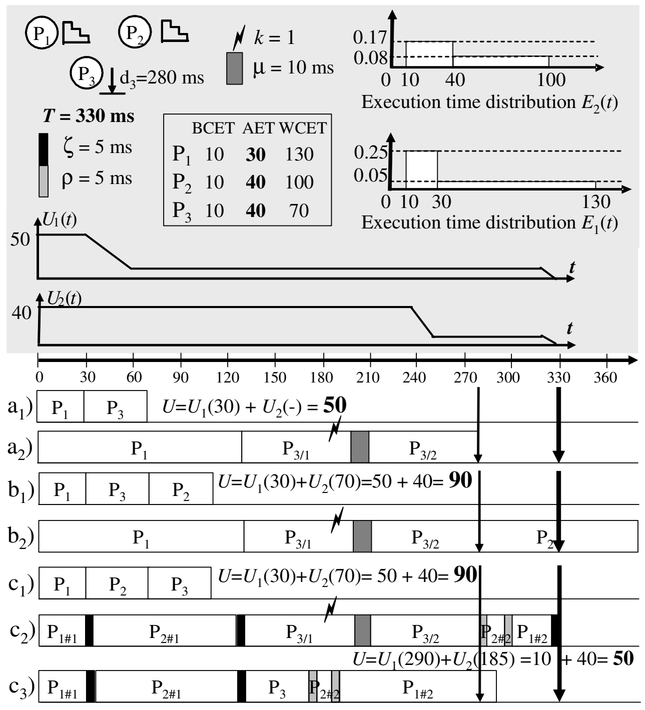


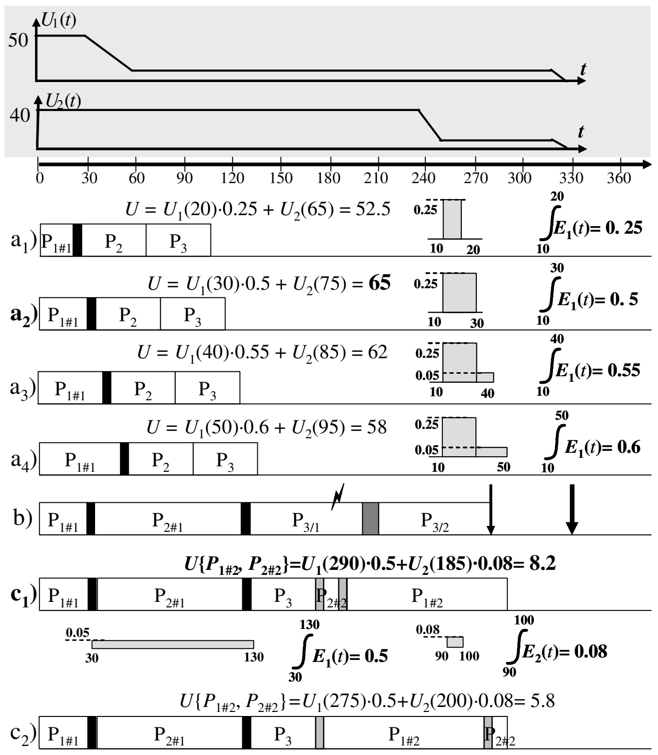
Figure 5. Example 2: Preemption vs. Dropping

the average case depicted in Fig. 5a$_1$. This schedule is valid in the worst-case, i.e., all deadlines are met, as shown in Fig. 5a$_2$, where P_1 and P_3 are executed with their worst-case execution times and hard process P_3 is affected by a fault and is re-executed. Note that we have chosen to schedule process P_1 before P_3 because, otherwise, if we schedule P_1 after P_3, the utility of such schedule in the average case will be only $U = U_1(70) + U_2(–) = 10 + 0 = 10$.

The static schedule with dropping generated for the average case can produce higher overall utility. However, such schedule is not safe. For example, in Fig. 5b$_1$, the schedule produces a high overall utility of 90 in the average case, but in the worst-case it will lead to deadline violations, as shown in Fig. 5b$_2$. Process P_2 exceeds the period deadline of 330 ms. A safety-critical application cannot be run with such a schedule.

In Fig. 5c, we present a flexible schedule that can be obtained with preemption applied on process executions. In the average case in Fig. 5c$_1$, the schedule produces the utility of 90. The worst-case is shown in Fig. 5c$_2$. All processes are executed with their worst-case execution times and process P_3 is re-executed. Processes P_1 and P_2 are preempted at 30 and 125 ms, respectively. After re-execution of process P_3, process P_2 is resurrected and completed until the end, while process P_1 is resurrected and again preempted before the end of the period. We account for preemption and resurrecting overheads, ζ and ρ, of 5 ms each. No deadlines are violated and the schedule is, hence, valid. Thus, the flexible schedule with preemption is able to produce an overall utility as high as the unsafe average-case schedule with dropping, while being, at the same time, safe.

The decision whether and where to preempt a process is the most crucial. In Fig. 6a, we present a reasoning of where to pre-



$U_1(t)$
50
$U_2(t)$
40
t
0 30 60 90 120 150 180 210 240 270 300 330 360
$U = U_1(20)\cdot 0.25 + U_2(65) = 52.5$
$U = U_1(30)\cdot 0.5 + U_2(75) = 65$
$U = U_1(40)\cdot 0.55 + U_2(85) = 62$
$U = U_1(50)\cdot 0.6 + U_2(95) = 58$
$\int E_1(t) = 0.25$
$\int E_1(t) = 0.5$
$\int E_1(t) = 0.55$
$\int E_1(t) = 0.6$
a1) a2) a3) a4) b) c1) c2)
$P_{1\#1}$ P_2 P_3
$P_{2\#1}$ $P_{3/1}$ $P_{3/2}$
$U\{P_{1\#2}, P_{2\#2}\}=U_1(290)\cdot 0.5+U_2(185)\cdot 0.08= 8.2$
$\int E_1(t)= 0.5$
$\int E_2(t)= 0.08$
$U\{P_{1\#2}, P_{2\#2}\}=U_1(275)\cdot 0.5+U_2(200)\cdot 0.08= 5.8$
$P_{1\#2}$ $P_{2\#2}$

Figure 6. Reasoning for Preemption and Resurrecting

empt process P_1. For example, in case in Fig. 6a$_1$, process P_1 is preempted at 20 ms, which corresponds to 25% of all its completion times (taking into account distribution of process P_1 execution time, see it on the right side of Fig. 6a$_1$). Thus, the utility of 50 will be produced in 25% of all the cases, which will contribute with the utility of 12.5 (i.e., 50×0.25). The overall utility of remaining soft processes, P_2 in our case, should be also taken into account because their utility will deviate with the different preemption. We consider the average-case execution times of the remaining processes in order to optimize the system for the average case. If process P_1 is preempted at 20 ms, process P_2 will complete in the average case at 65 ms, which contributes with the utility of 40. The total utility in this case will be $U = U_1(20) \times 0.25 + U_2(65) = 12.5 + 40 = 52.5$. If we preempt process P_1 at 30 ms, at its average execution time, as shown in Fig. 6a$_2$, it will contribute with the utility of 25 (50×0.5). Process P_2 will complete on average at 75 ms, which will again contribute with the utility of 40. The total utility is, hence, $U = U_1(30) \times 0.5 + U_2(75) = 25 + 40 = 65$. If we continue increasing preemption time for process P_1 with a step of 10 ms, we will obtain utility values of $U = U_1(40) \times 0.55 + U_2(85) = 22 + 40 = 62$ and $U = U_1(50) \times 0.6 + U_2(95) = 18 + 40 = 58$, respectively, as depicted in Fig. 6a$_3$ and Fig. 6a$_4$. Thus, the best preemption time for P_1 is 30 ms.

Process P_2 is also preempted in our schedule. However, the decision that process P_2 should be preempted at 125 ms was taken in order to guarantee the hard deadline of process P_3. As can be seen in Fig. 6b, re-execution of process P_3 completes directly before the deadline of 280 ms in the case that P_2 is preempted latest at 125 ms.

In the discussion so far we have ignored the value produced by the resurrected parts of the processes. For example, processes P_1 and P_2 will be resurrected after execution of process P_3 in Fig. 5c$_3$. Even though all processes are executed with their worst-case execution times, the overall utility is 50, which is as high as the utility in the best scenario of the pessimistic schedule with dropping. In Fig. 6c, we present a reasoning about resurrecting of processes P_1 and P_2. At first, we consider that process P_3 is executed with average execution time (since we optimize the schedule for the average case), while processes P_1 and P_2 are executed with their worst-case execution times because we want to know how much we can resurrect at maximum. There are two choices, depicted in Fig. 6c$_1$ and Fig. 6c$_2$. 100 ms of execution time of process P_1 and 10 ms of execution time of process P_2 are left, which correspond to 50% and 8% of process worst-case execution times, respectively, as depicted below Fig. 6c$_1$, where we consider execution time distributions for processes P_1 and P_2. The utility contribution with resurrected parts is 8.2 in Fig. 6c$_1$, where $P_{2\#2}$ is scheduled before $P_{1\#2}$, and is 5.8 in Fig. 6c$_2$, where $P_{1\#2}$ is scheduled before $P_{2\#2}$. Hence, we choose to schedule $P_{2\#2}$ before $P_{1\#2}$. Note that in the worst-case in Fig. 5c$_2$, we will have to preempt $P_{1\#2}$ at 325 ms to meet the period deadline of 330 ms (5 ms are accounted for the preemption overhead).

Any process, including its resurrected parts, can potentially be preempted at any time if this leads to a valid schedule with an increased utility. Any preempted process can be resurrected for increasing the overall utility if this does not lead to deadline violations. All preempted hard processes have to be always resurrected and have to complete before their deadlines even in case of faults. Our scheduling strategy, presented in Section 4, and the scheduling heuristics, presented in Section 5, are designed to explore preemption and resurrecting alternatives.

3.2 Static Scheduling vs. Quasi-Static Scheduling

Although the flexible single schedule with preemption can adapt to a particular situation, as was illustrated in Section 3.1, a quasi-static scheduling solution can further improve the produced overall utility. In [9] we have proposed a quasi-static scheduling for fault-tolerant embedded systems with soft and hard timing constraints. The main idea of quasi-static scheduling is to generate off-line a set of schedules, each explicitly generated for a particular situation that can happen online. These schedules will be available to an online scheduler, which will switch to the best one (the one that maximizes utility and guarantees the hard deadlines) depending on the occurrence of faults and the actual execution times of processes.

4. Problem Formulation

As an input we get an application $\mathcal{A}$, represented as a set of acyclic directed polar graphs $\mathcal{G}_k \in \mathcal{A}$, with a set $\mathcal{S}_k$ of soft processes and set $\mathcal{H}_k$ of hard processes. Soft processes are assigned with utility functions $U_i(t)$ and hard processes with hard deadlines d_i. Application $\mathcal{A}$ runs with a period T on a single computation node. The maximum number k of transient faults and the recovery overhead μ are given. The preemption overhead ζ and resurrecting overhead ρ are also given for each process (including hard processes). We know the best and worst-case execution times for each process, as presented in Section 2. The execution time distributions for all processes are given.

As an output, we have to obtain a quasi-static tree of schedules that maximizes the overall utility U of the application in the no-fault scenarios, maximizes the overall utility U^f in faulty scenarios, and satisfies all hard deadlines in all scenarios. The schedules are generated such that the utility (U and U^f, respectively) is maximized in the case that processes execute with the average-case execution times. We have to obtain the preemption time for each process. Resurrected parts of preempted processes have to be scheduled as well. It is important that the overall utility U of a no-fault scenario must not be compromised due to optimizing schedules for faulty scenarios. This is due to the fact that the no-fault scenario is the most likely to happen.

5. Scheduling Strategy and Algorithms

The goal of our scheduling strategy is to guarantee meeting the deadlines for hard processes, even in the case of faults, and to maximize the overall utility for soft processes. In addition, the utility of the no-fault scenario must not be compromised when building the fault-tolerant schedule because the no-fault scenario is the most likely to happen.

In this paper we will adapt a static scheduling strategy for hard processes, which we have proposed in [8] and which we have applied in [9] to generate fault-tolerant schedules for mixed soft and hard real-time systems. This strategy uses "recovery slack" in the schedule in order to accommodate time needed for re-executions in case of faults. After each process P_i we assign a slack equal to $(t_i^* + \mu) \times f$, where f is the number of faults to tolerate and t_i^* is the time allowed for the process to execute. $t_i^* = t_i^w$ if the process is not preempted. $t_i^* = t_i^{int} + \zeta$ if process P_i is preempted, where t_i^{int} is the execution time of P_i before the preemption. $t_i^* = \rho + t_i^{res}$ for resurrected parts of process P_i, where t_i^{res} is the execution time of the resurrected part. $t_i^* = \rho + t_i^{int\_res} + \zeta$ for resurrected parts being preempted, where $t_i^{int\_res}$ is the execution time of the resurrected part of process P_i before it is preempted. Note that both hard and soft processes can be preempted several times. The slack is shared between processes in order to reduce the time allocated for recovering from faults. In this paper, we will refer to such a fault-tolerant schedule with recovery slacks and preemption as an *if*-schedule.

```
SchedulingStrategy($\mathcal{G}$, $k$, $M$)
1  $S_{root}$ = FTSSP($\varnothing$, $\varnothing$, 0, $\mathcal{G}$, $k$)
2  if $S_{root} = \varnothing$ then return unschedulable
3  else
4      set $S_{root}$ as the root of fault-tolerant quasi-static tree $\Phi$
5      $\Phi$= FTQSP($\mathcal{G}$, $\Phi$, $S_{root}$, $k$, $M$)
6      return $\Phi$
7  end if
end SchedulingStrategy
```

Figure 7. General Scheduling Strategy

5.1 Scheduling Strategy

In our scheduling strategy, outlined in Fig. 7, we start by generating the *if*-schedule S_{root}, using the static scheduling algorithm for fault tolerance with preemption (FTSSP) presented in Section 5.2. The schedule is explicitly generated for the average case. The validity of the schedule, however, is investigated for the worst case, which guarantees that the schedule is safe. S_{root} contains the recovery slacks to tolerate k faults for hard processes and as many as possible faults for soft processes. The recovery slacks will be used by an online scheduler to re-execute processes online, without changing the order of process execution.

If the *if*-schedule S_{root} is not schedulable, i.e., one or more hard processes miss their deadlines, we conclude that the application is not schedulable and terminate. If the *if*-schedule S_{root} is schedulable, we generate the quasi-static tree Φ starting from schedule S_{root} by calling the FTQSP heuristic presented in Section 5.3, which uses FTSSP to generate safe *if*-schedules that maximize utility.

5.2 Static Scheduling Heuristics

Our static scheduling for fault tolerance and utility maximization with preemption (FTSSP), outlined in Fig. 8, is a list scheduling-based heuristic, which uses the concept of ready processes and ready list. By a "ready" process P_i we mean that all P_i's predecessors, required for starting process P_i, have been scheduled. The heuristic initializes the ready list R with processes ready at the beginning (line 2) and is looping while there is at least one process in the list (line 6).

In the case of synthesis of the root schedule $S = S_{root}$ (if $S_{parent} = \varnothing$), the algorithm sets the process time counter (CRT) to 0 and puts *all* the processes into the list U of unscheduled processes such that they can be executed with the worst-case execution times unless preempted.[1]

FTSSP addresses the problem of preempting and resurrecting of processes. All processes in the ready list R (both hard and soft) are evaluated if they need to be preempted (line 8) in order to satisfy deadlines. In the GetSafeIntr function, the evaluation where to preempt process P_i is done with schedule S_x composed of process P_i preempted at τ and only *unscheduled hard processes*. We evaluate each process P_i with evaluation step Δ from its earliest possible start time τ_{s,P_i}^b to its latest completion time τ_{c,P_i}^w.[2] If the execution of P_i until time τ leads to a deadline violation of any hard process or the schedule exceeds the system period, we conclude that P_i should be preempted at $\tau - \Delta$ time to meet the deadlines. We call this time moment a *forced preemption*. If the execution of entire or a certain part of process P_i leads to a schedulable solution, then process P_i is put into the list L of schedulable processes. Otherwise, it is removed from the

1. The initialization for synthesis of a schedule S inside the quasi-static tree (lines 1-4) will be discussed at the end of this section.
2. The evaluation step Δ is calculated as the average of average execution times of soft processes. The heuristic has been chosen based on the extensive experiments.

ready list R and is put into the stand-by list D (lines 10-13), which is initially set to $\varnothing$ (line 4).

After the list L of schedulable processes is created, the next step is to find which process out of the schedulable processes is the best to schedule first. We calculate priorities for *all* unscheduled soft processes using the MU priority function presented in [4] (line 16). The MU priority function computes for each soft process P_i the value, which constitutes of the utility produced by P_i, scheduled as early as possible, and the sum of utility contributions of the other soft processes delayed because of P_i. The utility contribution of the delayed soft process P_j is obtained as if process P_j would complete at time $t_j = (t_j^E + t_j^L) / 2$. t_j^E and t_j^L are the completion times of process P_j in the case that P_j is scheduled after P_i as early as possible and as

```
FTSSP($P_s$, $S_{parent}$, T, $k$, $\mathcal{G}$)
1  $S \rightarrow parent = S_{parent}$
2  $R$ = GetReadyNodes($P_s$, $S_{parent}$, $\mathcal{G}$); $CRT$ = T
3  $U$ = GetUnschedulable($P_s$, $S_{parent}$, $\mathcal{G}$)
4  $D = \varnothing$;  CalculateExecTimes($S_{parent}$, $U$);
5  $\Delta$ = ObtainEvaluationStep($U$)
6  while $R \neq \varnothing$ do
7    for all $P_i \in R$ do
8      $\tau_{forced}$ = GetSafeIntr($P_i$, $U$, $\Delta$)
9      if $\tau_{forced} > \tau_s^b$ then $L = L \cup P_i$
10     else
11       Remove($R$, $P_i$)
12       $D = D \cup P_i$
13     end if
14   end for
15   if $L \neq \varnothing$ then
16     CalculatePriorities($U$)
17     $P_{best}$ = GetBestProcess($L$)
18     $\tau_{best}$ = GetBestIntr($P_{best}$, $U$, $\Delta$)
19     if $\tau_{best} = \tau_s^b$ then
20       $D = D \cup P_{best}$
21     else if $\tau_{best} < \tau_c^w$ then
22       Schedule($S$, $CRT$, $P_{best}$, $\tau_{best}$)
23       AddRecoverySlack($P_{best}$, $\tau_{best}$, $U$)
24       SubstractExecTime($P_{best}$, $\tau_{best}$)
25       $R = R \cup D$;$D = \varnothing$; $D = D \cup P_{best}$
26     else
27       Schedule($S$, $CRT$, $P_{best}$, $\tau_{c,Pbest}^w$)
28       AddRecoverySlack($P_{best}$, $\tau_{c,Pbest}^w$, $U$)
29       $R = R \cup D$; $D = \varnothing$
30       Remove($U$, $P_{best}$)
31       AddSuccessors($R$, $P_{best}$)
32     end if
33     Remove($R$, $P_{best}$)
34   end if
35   while $R = \varnothing$ and $D \neq \varnothing$ do
36     $H$ = GetSchedulableHardProcesses($D$)
37     if $H \neq 0$ then
38       $P_H$ = GetBestProcess($H$)
39       Schedule($S$, $CRT$, $P_H$, $\tau_{c,PH}^w$)
40       AddRecoverySlack($P_{best}$, $\tau_{c,PH}^w$, $U$)
41       Remove($D$, $P_H$);Remove($U$, $P_H$)
42       $R = R \cup D$; $D = \varnothing$
43       AddSuccessors($R$, $P_H$)
44     else
45       $P_S$ = GetBestToDrop($D$)
46       if $P_S = \varnothing$ and $H = \varnothing$ then return unschedulable
47       Remove($D$, $P_S$);Remove($U$, $P_S$)
48       AddSuccessors($R$, $P_S$)
49     end if
50   end while
51 end while
52 return $S$
end FTSSP
```

Figure 8. Static Scheduling with Preemption

late as possible, respectively. The GetBestProcess function (line 17) selects either the soft process P_{best} with the highest MU priority or, if there are no soft processes in the ready list, the hard process P_{best} with the earliest deadline.

Once the process P_{best} is selected, the algorithm evaluates with the GetBestIntr heuristic (line 18) if it should be preempted in order to increase the overall utility value. We check all the valid execution times, e.g. from the earliest possible start time τ_s^b until the forced preemption, with the evaluation step Δ. However, to determine exactly whether process P_{best} should be preempted at τ, we should consider all possible combinations of preemption for the remaining unscheduled processes and choose the best-possible combination. This is infeasible for large applications. Instead, we use a preemption evaluation heuristic, where we generate a schedule S_y, which is composed of process P_{best} preempted at τ and only *unscheduled soft processes*. The selection of process order in the schedule is done based on the MU priority function as in the main algorithm. If P_{best} is a soft process, its remaining part will be also placed into the schedule S_y. However, at least one process has to be placed into the schedule between the preempted part and the remaining part of P_{best}. The obtained overall utility of the schedule S_y will indicate the contribution of preempting process P_{best} at time τ. We choose the preemption time τ_{best} that produces the best utility.

Depending on the value of τ_{best}, two other options are possible besides preempting of a process: postponing the process if $\tau_{best} = \tau_s^b$ or full execution if $\tau_{best} = \tau_c^w$, where τ_s^b is earliest possible start time and τ_c^w is latest completion time of process P_{best}.

If process P_{best} is postponed (lines 19-20), it is put into the stand-by list D and will be allowed to be selected only if at least one process has been scheduled. If process P_{best} is preempted (lines 21-25), its remaining part is also placed into the stand-by list D under the same condition as a postponed process. Then the process is scheduled until best time to preempt τ_{best}. Its execution time is subtracted to account for scheduled preempted part (line 24). In case of $\tau_{best} = \tau_c^w$ (lines 27-31), process P_{best} will be completely scheduled with its full execution time and its successors will be put into the ready list R.

In the case that the process P_{best} is fully scheduled or in the case that a part of process P_{best} is scheduled, the heuristic copies the processes from the stand-by list D to the ready list R and empties the stand-by list D (lines 25 and 29). If P_{best} has been preempted, then the stand-by list D will contain only the remaining part of process P_{best} (line 25).

Once process P_{best} is scheduled (lines 22 or 27), the recovery slack with the number of re-execution is assigned to it with the AddRecoverySlack heuristic (lines 23 or 28). For a hard process, we always assign k re-executions. If P_{best} is a soft process, then the number of re-executions has to be calculated. First, we compute how many times P_{best} can be re-executed without violating deadlines. We schedule P_{best}'s re-executions one-by-one directly after P_{best} and check schedulability (by generating S_x-schedules). If the re-execution is schedulable, we evaluate if it is better to drop the re-execu-

```
FTQSP($\mathcal{G}$, $\Phi$, $S_{root}$, $k$, $M$)
1  $S = S_{root}$
2  while $size(\Phi) < M$ do
3    $\Phi_0 = \varnothing$
4    for all $P_i \in S$ do
5      $\tau_{c,P_i}{}^b$ = GetBestCompTime($S$, $P_i$)
6      $\tau_{c,P_i}{}^w$ = GetWorstCompTime($S$, $P_i$)
7      $S_{opt}$ = FTSSP($S$, $P_i$, $\tau_{c,P_i}{}^b$, $k$, $\mathcal{G}$)
8      $S_{pess}$ = FTSSP($S$, $P_i$, $\tau_{c,P_i}{}^w$, $k$, $\mathcal{G}$)
9      if $S_{opt} \neq \varnothing$ and $S_{pess} \neq \varnothing$ then
10       $S\{P_i\} \rightarrow switch\_pts$ = IntervalPartitioning($S_{opt}$, $S_{pess}$, $\tau_{c,P_i}{}^b$, $\tau_{c,P_i}{}^w$)
11       $\Phi_0 = \Phi_0 \cup S_{opt}$
12       $\Phi_0 = \Phi_0 \cup S_{pess}$
13     end if
14   end for
15   $\Phi = \Phi \cup$ GetBestSchedules($\Phi_0$, $M$)
16   $S$ = SelectNextSchedule($\Phi$)
17 end while
18 return $\Phi$
end FTQSP
```

Figure 9. Quasi-Static Scheduling

tion for maximizing the overall utility U^f of this particular fault scenario (by generating S_y-schedules).

Process P_{best} is removed from the ready list R after being scheduled or postponed (line 33) and the algorithm continues from the beginning except the case that all processes are in the stand-by list D, while the ready list R is empty (line 35). This can happen after several iterations of extensive postponing. To handle this situation, we first create a list H of schedulable hard processes from D (line 36). If there exists at least one (schedulable) hard process in H, the GetBestProcess function selects from H the hard process P_H with the closest deadline (line 38). The hard process P_H is then scheduled with its full execution time, assigned with recovery slack, it is removed from the ready list R and the list U of unschedulable processes, and its successors are put into the ready list (lines 39-43). If the list H of schedulable hard processes is empty, then we look for a soft process P_S in the stand-by list D, which can be removed from the system with the lowest degradation of the overall utility (line 45). If no soft process P_S is found and the list H is empty, we conclude that the system is unschedulable (line 46). Otherwise, we *drop* the soft process P_S by removing it from the stand-by list D and the list U of unschedulable processes, and add its successors into the ready list R (lines 47-48). In such case, process P_S will not have a chance to be scheduled and is actually *dropped*.

FTSSP returns an *if*-schedule S explicitly generated for the average case providing a high overall utility (line 53). The return schedule is also guaranteed to satisfy hard deadlines in the worst case, i.e., the schedule is safe.

In the case of synthesis of a schedule S inside the quasi-static tree, the FTSSP heuristic will be called from the quasi-static scheduling (FTQSP) algorithm discussed in the next section. An online scheduler will switch on such schedule S upon completion time T of process P_s from the schedule S_{parent}. FTSSP will initially set S_{parent} as a parent for S (line 1), set the *CRT* counter to T (line 2), and the list U of unscheduled processes will contain all not completed processes in schedule S_{parent} (line 3). The processes, which have not started, can be executed in schedule S with their worst-case execution times. The processes, which have been preempted in S_{parent}, can complete their execution in S and can be executed with their remaining execution times (line 4). These constraints are captured in all our scheduling steps in FTSSP and the synthesis of schedule S inside the quasi-static tree is then performed exactly as discussed above for the root schedule.

5.3 Quasi-Static Scheduling Heuristic

In general, quasi-static scheduling should generate a tree of schedules that will adapt to different execution situations. However, tracing all execution scenarios is infeasible. Therefore, we reduce the number of schedules in the quasi-static tree Φ by considering only the best-case and the worst-case completion times of processes in each particular schedule. We have adapted our quasi-static scheduling algorithm from [9] to explore different schedules with preemption and resurrected processes. The *if*-schedules are generated with the static scheduling heuristic FTSSP discussed above.

The quasi-static scheduling algorithm is outlined in Fig. 9. The algorithm takes as an input the process graph $\mathcal{G}$, the initial quasi-static tree Φ, which contains only the root *if*-schedule, the root *if*-schedule S_{root}, the maximum number k of faults, and the maximum number M of schedules in the tree. The current *if*-schedule S is initialized with the root *if*-schedule S_{root} (line 1). The heuristic is looping until M *if*-schedules have been generated (line 2).

For each process P_i in the schedule S we generate optimistic and pessimistic *if*-schedules with FTSSP (lines 5-8), where the first one corresponds to the best-case completion time $\tau_{c,P_i}{}^b$ of process P_i and the second one to the worst-case completion time $\tau_{c,P_i}{}^w$ of process P_i. Note that completion times $\tau_{c,P_i}{}^b$ and $\tau_{c,P_i}{}^w$ correspond to the no faulty scenario in *if*-schedule S. If both schedules are valid (line 9), the interval between $\tau_{c,P_i}{}^b$ and $\tau_{c,P_i}{}^w$ is traced on the *interval-partitioning step* (line 10). For completion time $\tau_{c,P_i}{}^j$ either the pessimistic schedule or the optimistic schedule is assigned according to the overall utility produced by the schedule, i.e., if the utility of the optimistic schedule is better than the utility of the pessimistic schedule at completion $\tau_{c,P_i}{}^j$, then if process P_i completes at time $\tau_{c,P_i}{}^j$, the optimistic schedule will be chosen. The optimistic schedule is not always safe, therefore, if there exists a risk of violating deadlines, the pessimistic schedule, which is always safe, will be chosen. For example, if the optimistic schedule may lead to deadline violations at completion time $\tau_{c,P_i}{}^j$, the pessimistic schedule will be executed if process P_i completes at $\tau_{c,P_i}{}^j$. The completion times of process P_i, at which switching between schedules should be done, are assigned to the process P_i in the schedule S. The optimistic and pessimistic schedules are stored in the temporary set of schedules Φ_0 (lines 11-12).

Out of the pessimistic and optimistic *if*-schedules in Φ_0, we store in the quasi-static tree Φ only such *if*-schedules that, if switched to, lead to the most significant improvement in terms of the overall utility (line 15). Afterwards, the new best *if*-schedule S is selected in the tree Φ (line 16). For all processes in S we will perform the same procedure as described above on the

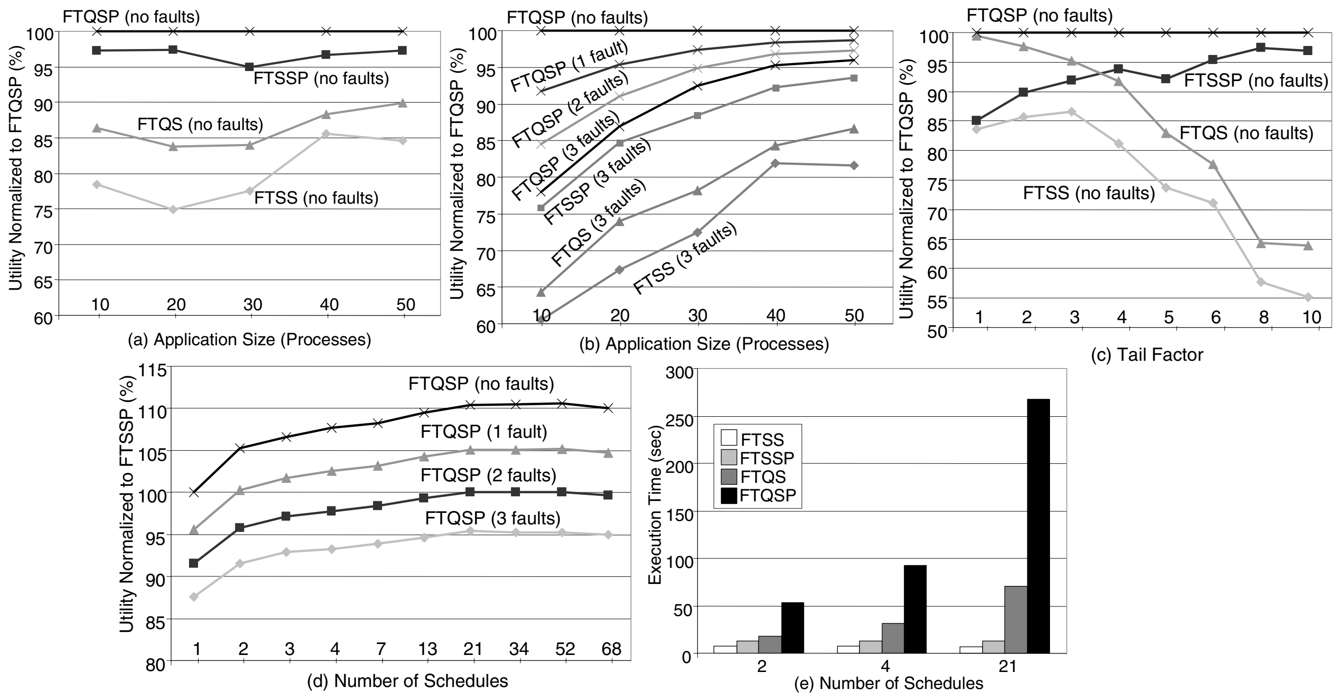


Figure 10. Experimental Results

next loop iteration, i.e., generate optimistic and pessimistic *if*-schedules and select the ones that, eventually, will contribute to the most significant improvement if switched to.

6. Experimental Results

For the experiments, we have generated 100 applications with 10, 20, 30, 40, and 50 processes, where we have varied average case execution times (AETs) between 1 and 100 ms, the best-case execution time (BCETs) between 0 ms and the average case execution times. The worst-case execution times (WCETs) have been assigned to capture the effect of "much larger execution times in the worst-case", e.g. the effect of *tails*. We have associated with each process P_i at every application a *tail factor* $TF_i = AET_i \times 2 / WCET_i$. The tail factor has been randomly generated between 1 and 10. Thus, we have calculated the worst-case execution times as $WCET_i = TF_i \times AET_i \times 2$. We have set 75% of all processes as soft and have associated pre-generated step utility functions to them. The other 25% of processes have been associated with the local hard deadlines. The number of transient faults have been set to $k = 3$. The recovery overhead μ, the resurrecting overhead ρ and the preemption overhead ζ have been randomly generated for every process P_i at every application between 1 and 30 per cent of P_i's average-case execution time, rounded to the greatest integer value. The experiments have been run on a Pentium 4 2.8 GHz processor with 1Gb memory.

In the first set of experiments, we have evaluated the improvement that can be obtained with our novel fault tolerance static and quasi-static scheduling with preemption compared to the fault tolerance static and quasi-static scheduling without preemption, which we have proposed in [9]. Thus, we have evaluated four algorithms:

- the static scheduling algorithm with preemption, dropping and fault tolerance (FTSSP), proposed in Section 5.2;
- the quasi-static scheduling algorithm with preemption, dropping and fault tolerance (FTQSP), proposed in Section 5.3, which uses FTSSP to generate the schedules in the quasi-static tree;
- the static scheduling algorithm with dropping and fault tolerance but without preemption (FTSS) from [9];
- the quasi-static scheduling algorithm with dropping and fault tolerance but without preemption (FTQS) from [9], which uses FTSS to generate the schedules in the quasi-static tree.

In Fig. 10a, we depict the utilities produced in the case of no faults with the schedules generated with four algorithms for the applications composed of 10, 20, 30, 40 and 50 processes. The utilities are normalized to the utility produced in case of no faults by the schedules generated with FTQSP. As can be seen in Fig. 10a, FTQSP generates the best schedules and the scheduling algorithms with preemption outperform the scheduling algorithms without preemption. FTQSP is better than FTQS by 10-15%. FTSSP is better than FTSS by 15-20%. Thus, preemption plays an important role during generation of schedules.

In Fig. 10b, we present the reduction of quality of schedules produced with FTQSP with the number of faults. The quality of the FTQSP schedules in case of 1 fault degrades with 7% for 10 processes and with 2% for 50 processes. In case of 3 faults, the quality degrades with 22% for 10 processes while with only 6% for 50 processes. However, in case of 3 faults, the FTQSP schedules are better than the FTQS schedules by 15%. FTSSP is better than FTQS by more than 10% and is better than FTSS by approximately 20%. Thus, even in the case of faults, preemption is important.

As can be seen in Fig. 10a and Fig. 10b, the quasi-static scheduling algorithm with preemption, FTQSP, is better than the static scheduling algorithm with preemption, FTSSP, only by 3-5%. However, if we reduce the tail factor from 10 to 2, as illustrated in Fig. 10c, FTQSP becomes better than FTSSP with already 10%, and, if we reduce the tail factor to 1, FTQSP is better than FTSSP with 15%.

As the tail factor increases, the efficiency of the scheduling heuristics with preemption, as such, increases. In the case of tail factor 1, FTQSP is better than FTQS by only 1% and FTSSP is better than FTSS by only 2%. However, in the case of tail factor 10, FTQSP outperforms FTQS with 36% and FTSSP outperforms FTSS with 42%.

In the other set of experiments, presented in Fig. 10d, we evaluate how many schedules need to be generated in order to obtain a substantial improvement of FTQSP over FTSSP. The experiments have been run in the case of tail factor 2 for applications composed of 20 processes. The utility produced with the schedule generated with FTSSP in the case of no faults has been chosen as a baseline. We depict the normalized utilities of schedules produced with FTQSP in the case of no faults, 1 fault, 2 faults, and 3 faults. The saturation point is in 21 schedules, where the improvement is 11%. However, with only 4 schedules the improvement of FTQSP is already 8%. In other words, there is no need to generate many schedules with FTQSP in order to improve a one-schedule FTSSP solution. The execution times of the quasi-static heuristics for 2, 4 and 21 schedules are shown in Fig. 10e. FTQSP is approximately three times slower than FTQS. We also show, for reference, the execution times of the FTSSP and FTSS heuristics, which generate a single schedule. FTSSP is two times slower than FTSS.

We have also run our experiments on a real-life example, a vehicle cruise controller (CC) composed of 32 processes [10], which is implemented on a single microcontroller with a memory unit and communication interface. 16 processes, which are critically involved with the actuators, have been considered hard. We have set $k = 3$ and have considered μ, ζ, and ρ between 1 and 30% of process average-case execution times. The tail factor has been set to 10. The quasi-static scheduling algorithm with preemption, FTQSP, generates schedules that outperform the schedules produced with the quasi-static scheduling algorithm without preemption, FTQS, with 18% in case of no faults, 1 fault, and 2 faults, and with 17% in case of 3 faults (in terms of utility).

The scheduling approaches with preemption are able to produce schedules with substantially better quality than the approaches without preemption. A quasi-static tree of schedules should be generated, not a single schedule, in order to satisfy timing constraints and generate the high utility independent of distribution of process execution times.

7. Conclusions

In this paper we have presented the quasi-static scheduling approach for generation of fault-tolerant schedules with preemption of process executions. Schedules with preemption are generated off-line and are adaptable to the situations that can happen on-line during execution of the application such as fault occurrences, overloading, long process executions. The schedules maximize the overall utility of application in the average-case execution scenarios while preserving hard timing constraints in all possible scenarios.

Our experimental results have shown the advantage of using preemption for fault-tolerant embedded systems with hard and soft timing constraints compared to the previous scheduling approaches. Preemption is essential for generating schedulable and fault-tolerant solutions with the high overall utility.

8. References

[1] L. Abeni and G. Buttazzo, “Integrating Multimedia Applications in Hard Real-Time Systems”, *RTSS*, 4-13, 1998.

[2] H. Aydin, R. Melhem, and D. Mosse, “Tolerating Faults while Maximizing Reward”, *12th Euromicro Conf. on RTS*, 219–226, 2000.

[3] G. Buttazzo and F. Sensini, “Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments”, *IEEE Trans. on Computers*, 48(10), 1035–1052, 1999.

[4] L.A. Cortes, P. Eles, and Z. Peng, “Quasi-Static Scheduling for Real-Time Systems with Hard and Soft Tasks”, *DATE Conf.*, 1176-1181, 2004.

[5] R. I. Davis, K. W. Tindell, and A. Burns, “Scheduling Slack Time in Fixed Priority Pre-emptive Systems”, *RTSS*, 222–231, 1993.

[6] C. Dima, A. Girault, C. Lavarenne, and Y. Sorel, “Off-line Real-Time Fault-Tolerant Scheduling”, *Euromicro Parallel and Distributed Processing Workshop* , 410–417, 2001.

[7] C. C. Han, K. G. Shin, and J. Wu, “A Fault-Tolerant Scheduling Algorithm for Real-Time Periodic Tasks with Possible Software Faults”, *IEEE Trans. on Computers*, 52(3), 362–372, 2003.

[8] V. Izosimov, P. Pop, P. Eles, and Z. Peng, “Design Optimization of Time- and Cost-Constrained Fault-Tolerant Distributed Embedded Systems”, *DATE Conf.*, 864-869, 2005.

[9] V. Izosimov, P. Pop, P. Eles, and Z. Peng, “Scheduling of Fault-Tolerant Embedded Systems with Soft and Hard Timing Constraints”, *DATE Conf.*, 2008.

[10]V. Izosimov, “Scheduling and Optimization of Fault-Tolerant Embedded Systems”, *Licentiate Thesis No. 1277, Dept. of Computer and Information Science, Linköping University,* 2006.

[11]N. Kandasamy, J. P. Hayes, and B. T. Murray, “Transparent Recovery from Intermittent Faults in Time-Triggered Distributed Systems”, *IEEE Trans. on Computers*, 52(2), 113–125, 2003.

[12]H. Kopetz, “Real-Time Systems - Design Principles for Distributed Embedded Applications”, *Kluwer Academic Publishers*, 1997.

[13]F. Liberato, R. Melhem, and D. Mosse, “Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard Real-Time Systems”, *IEEE Trans. on Computers*, 49(9), 906–914, 2000.

[14]P.M. Melliar-Smith, L.E. Moser, V. Kalogeraki, and P. Narasimhan, “Realize: Resource Management for Soft Real-Time Distributed Systems”, *DARPA Information Survivability Conf.*, 1, 281–293, 2000.

[15]C. Pinello, L. P. Carloni, A. L. Sangiovanni-Vincentelli, “Fault-Tolerant Deployment of Embedded Software for Cost-Sensitive Real-Time Feedback-Control Applications”, *DATE*, 1164–1169, 2004.

[16]Wang Fuxing, K. Ramamritham, and J.A. Stankovic, “Determining Redundancy Levels for Fault Tolerant Real-Time Systems”, *IEEE Trans. on Computers*, 44(2), 292–301, 1995.

[17]Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, and M.J. Irwin, “Reliability-Aware Co-synthesis for Embedded Systems”, *Proc. 15th IEEE Intl. Conf. on Appl.-Spec. Syst., Arch. and Proc.*, 41–50, 2004.

[18]Ying Zhang and K. Chakrabarty, “A Unified Approach for Fault Tolerance and Dynamic Power Management in Fixed-Priority Real-Time Embedded Systems”, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 25(1), 111–125, 2006.