
Izosimov, Viacheslav; Pop, Paul; Eles, Petru; Peng, Zebo

Published in:
11th Euromicro Conference on Digital System Design

Link to article, DOI:
10.1109/DSD.2008.47

Publication date:
2008

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights. Users may download and print one copy of any publication from the public portal for the purpose of private study or research.

- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

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 faults1, 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] a 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 \( A \) as a set of directed, acyclic, polar graphs \( G_i = (V_i, E_i) \) \( \in A \). Each node \( P_i \in V_i \) represents one process. An edge \( e_{ij} \in E_i \) 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 \( P_i \), \( t_{i}^a \), 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 \( A \) consisting of the process graph \( G_1 \) with three processes, \( P_1, P_2 \) and \( P_3 \). The execution times for the processes are shown in the table. \( \mu \) is a recovery overhead, which represents the time needed to start re-execution of a process in case of faults. \( \zeta \) is a preemption overhead, which represents the time needed to preempt a process and store its state (to “checkpoint”). \( \rho \) 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 \( G_i \) have the same period \( T = T_G \), which is the period of the process graph. In Fig. 1 process graph \( G_1 \) 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 \( P_3 \) is assigned with a utility function \( U_3(t) \) 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, \( P_1 \) and \( P_2 \), is 25, in the case that \( P_1 \) completes at 50 ms and \( P_2 \) 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 \( A \) from Fig. 1 is run with preemption. A hard process \( P_1 \), even if preempted, has to always complete its execution before the deadline and, thus, has to be always resurrected. We will denote with \( P_i \) the execution of \( j \)th 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 \( P_1 \) was preempted, its utility is 0 unless the process is resurrected, i.e., \( U_1(t) = 0 \). In Fig. 2c, processes \( P_1 \) and \( P_2 \) produce utility of 0 after they are preempted at 30 ms and 65 ms. If \( P_1 \) is resurrected and completed at time \( t \), then its utility is calculated according to its utility function \( U_1(t) \). 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
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 \( P_1 \) completes before it is preempted, it will produce utility \( U(t) \). 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., \( U(-) = 0 \). In the execution in Fig. 2e we drop process \( P_2 \) of application \( A \). Thus, process \( P_1 \) completes at 50 ms and process \( P_3 \) at 110 ms, which gives the total utility \( U = U_1(50) + U_2(110) = 10 + 0 = 10 \).

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 \( P_j \) 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 \( \alpha_i \) represents the stale value coefficient. \( \alpha_i \) 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:

\[ \alpha_i = \frac{1 + \sum_{j \in DP(P_i)} \alpha_j}{1 + |DP(P_i)|} \]

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.

Suppose that soft process \( P_2 \) has two predecessors, \( P_1 \) and \( P_3 \). If \( P_1 \) is preempted while \( P_2 \) and \( P_3 \) are completed successfully, then, according to the formula, \( \alpha_i = (1 + 0 + 1) / (1 + 2) = 7/6 \). Hence, \( U_2(t) = 7/6 \times U_2(t) \). The use of a stale value will propagate though the application. For example, if soft process \( P_3 \) is the only successor of \( P_3 \), and is completed, then \( \alpha_3 = (1 + 2 + 2) / (1 + 1) = 5/6 \). Hence, \( U_3(t) = 5/6 \times U_3(t) \).

### 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.

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 ex-
In the average case, the schedule will contain both processes with their worst-case execution times and process $P_2$ is re-executed, $P_1$’s execution will lead to missed 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. 4a1, 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. 4a2, 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 $\mu$ 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 $U = U_1(30) + U_2(\cdot) = 50 + 0 = 50$ in the average case depicted in Fig. 5a1. This schedule is valid in the worst-case, i.e., all deadlines are met, as shown in Fig. 5a3, where $P_1$ and $P_2$ 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 $U = U_1(70) + U_2(\cdot) = 10 + 0 = 10$.

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. 5b1, 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. 5b2. 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. 5c1, the schedule produces the utility of 90. The worst-case is shown in Fig. 5c2. 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, $\zeta$ and $\rho$, 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-
empt process $P_1$. For example, in case in Fig. 6a1, 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. 6a1). 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. 6a2, 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. 6a3 and Fig. 6a4. 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 $P_2$ 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. 5c3. 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 $P_1$ 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. 6c1 and Fig. 6c2. 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. 6c1, where we consider execution time distributions for processes $P_1$ and $P_2$. The utility contribution with resurrected parts is 8.2 in Fig. 6c2, where $P_{2\text{res}}$ is scheduled before $P_{1\text{res}}$ and is 5.8 in Fig. 6c2, where $P_{1\text{res}}$ is scheduled before $P_{2\text{res}}$. Hence, we choose to schedule $P_{2\text{res}}$ before $P_{1\text{res}}$. Note that in the worst-case in Fig. 5c2, we will have to preempt $P_{1\text{res}}$ 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 $A$, represented as a set of acyclic directed polar graphs $G \in A$, with a set $S$ of soft processes and set $H$ of hard processes. Soft processes are assigned with utility functions $U(t)$ and hard processes with hard deadlines $d_t$. Application $A$ runs with a period $T$ on a single compu-
tation node. The maximum number $k$ of transient faults and the recovery overhead $\mu$ are given. The preemption overhead $\zeta$ and resurrecting overhead $\rho$ 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 $(t_i^* + \mu) \times f_i$, where $f_i$ 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, $t_i^{res}$ is the execution time of the resurrected parts of process $P_i$, where $t_i^{res}$ is the execution time of the resurrected part, $t_i^* = \rho + t_i^{int,rel} + \zeta$ for resurrected parts being preempted, where $t_i^{int,rel}$ 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.

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 $\Phi$ 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 $S_{parent} = \emptyset$), 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.

$FTSSP$ 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 $S_i$ composed of process $P_j$ preempted at $\tau$ and only unscheduled hard processes. We evaluate each process $P_i$ with evaluation step $\Delta$ from its earliest possible start time $\tau_{e,P_i}$ to its latest completion time $\tau_{c,P_i}$. If the execution of $P_i$ until time $\tau$ 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 $\Delta$ 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 $\emptyset$ (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 $t_j = (t_j^E + t_j^F)$ / 2. $t_j^E$ and $t_j^F$ 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 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 $\tau_{best}$ until the forced preemption, with the evaluation step $\Delta$. However, to determine exactly whether process $P_{best}$ should be preempted at $\tau$, we should consider all possible combinations of preemption for the remaining unschedulable 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 $\tau$ and only unschedulable 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 $\tau$. We choose the preemption time $\tau_{best}$ that produces the best utility.

Depending on the value of $\tau_{best}$ 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 $\tau_s^b$ is earliest possible start time and $\tau_c^w$ 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 $\tau_{best}$. 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, 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-

FTSSP($P_x$, $S_{parent}$, $T$, $k$, $\phi$)

1. $S \rightarrow parent = S_{parent}$
2. $R = GetReadyNodes($$P_x$, $S_{parent}$, $\phi$); $CRT = T$
3. $U = GetUnscheduleable($$P_x$, $S_{parent}$, $\phi$)
4. $D = \emptyset$; CalculateExecTimes($S_{parent}$, $U$)
5. $\Delta = ObtainEvaluationStep($$U$)
6. while $R \neq \emptyset$ do
7. for all $P_i \in R$ do
8. \hspace{1em} $\tau_{forced} = GetSafeIntr($$P_i$, $U$, $\Delta$)
9. \hspace{1em} if $\tau_{forced} > t^w_i$ then $L = L \cup P_i$
10. \hspace{1em} else\hspace{1em} Remove($R$, $P_i$)
11. \hspace{1em} $D = D \cup P_i$
12. end if
13. end for
14. if $L = \emptyset$ then
15. CalculatePriorities($U$)
16. $P_{best} = GetBestProcess($$U$)
17. $\tau_{best} = GetBestIntr($$P_{best}$, $U$, $\Delta$)
18. if $\tau_{best} < \tau_{c}\Delta^m$ then
19. \hspace{1em} Schedule($S$, $CRT$, $P_{best}$, $\tau_{best}$)
20. \hspace{1em} AddRecoverySlack($P_{best}$, $\tau_{c}\Delta^m$, $U$)
21. \hspace{1em} SubtaskExecTime($P_{best}$, $\tau_{best}$)
22. \hspace{1em} $R = R \cup D; D = \emptyset$
23. end if
24. end if
25. Schedule($S$, $CRT$, $P_{best}$, $\tau_{c}\Delta^m$)
26. AddRecoverySlack($P_{best}$, $\tau_{c}\Delta^m$, $U$)
27. $R = R \cup D; D = \emptyset$
28. Remove($U$, $P_{best}$)
29. AddSuccessors($R$, $P_{best}$)
30. end if
31. end if
32. while $R \neq \emptyset$ and $D \neq \emptyset$ do
33. $H = GetScheduledHardProcesses($$D$)
34. if $H \neq \emptyset$ then
35. \hspace{1em} $P_y = GetBestProcess($$H$)
36. \hspace{1em} Schedule($S$, $CRT$, $P_y$, $\tau_{c}PH^m$)
37. \hspace{1em} AddRecoverySlack($P_y$, $\tau_{c}PH^m$, $U$)
38. \hspace{1em} Remove($D$, $P_y$), Remove($U$, $P_y$)
39. \hspace{1em} $R = R \cup D; D = \emptyset$
40. \hspace{1em} AddSuccessors($R$, $P_y$)
41. \hspace{1em} else\hspace{1em}$P_y = GetBestToDrop($$D$)
42. \hspace{1em} if $P_y = \emptyset$ and $H = \emptyset$ then return unschedulable
43. \hspace{1em} Remove($D$, $P_y$), Remove($U$, $P_y$)
44. \hspace{1em} AddSuccessors($R$, $P_y$)
45. \hspace{1em} end if
46. \hspace{1em} end if
47. end while
48. return $S$
49. end FTSSP
FTQSP(\(G \Phi S_{root}, k, M\))
1 \(S = S_{root}\)
2 while size(\(q\)) < M do
3 \(q_{b} = \emptyset\)
4 for all \(P_{i} \in S\) do
5 \(\tau_{c,P_{i}} = GetBestCompTime(S, P_{i})\)
6 \(\tau_{c,P_{i}} = GetWorstCompTime(S, P_{i})\)
7 \(S_{opt} = FTSSP(S, P_{i}, \tau_{c,P_{i}}, k, g)\)
8 \(S_{pass} = FTSSP(S, P_{i}, \tau_{c,P_{i}}^{v}, k, g)\)
9 if \(S_{opt} \neq \emptyset\) and \(S_{pass} \neq \emptyset\) then
10 \(S[\{P_{j}\}] = switch\_pils = IntervalPartitioning(S_{opt}, S_{pass}, \tau_{c,P_{i}^{v}}, \tau_{c,P_{i}}^{v})\)
11 \(q_{b} = q_{b} \cup S_{opt}\)
12 \(q_{b} = q_{b} \cup S_{pass}\)
13 end if
14 end for
15 \(\Phi = \Phi \cup GetBestSchedules(q_{b}, M)\)
16 \(S = SelectNextSchedule(\Phi)\)
17 end while
18 return \(\Phi\)
end FTQSP

Figure 9. Quasi-Static Scheduling

1. Introduction for maximizing the overall utility \(U^{f}\) of this particular fault scenario (by generating \(S_{r}\)-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 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 within 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 \(P_{i}\) 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 unschedulable 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 \(\Phi\) 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 \(G\) the initial quasi-static tree \(\Phi\) 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. 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 \(\tau_{c,P_{i}}^{l}\) 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 \(\tau_{c,P_{i}}^{l}\), then if process \(P_{i}\) completes at time \(\tau_{c,P_{i}}^{l}\), 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 \(\tau_{c,P_{i}}^{l}\), the pessimistic schedule will be executed if process \(P_{i}\) completes at \(\tau_{c,P_{i}}^{l}\). 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 \(\Phi_{0}\) (lines 11-12).

Out of the pessimistic and optimistic \(if\)-schedules in \(\Phi_{0}\) we store in the quasi-static tree \(\Phi\) 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 \(\Phi\) (line 16). For all processes in \(S\) we will perform the same procedure as described above on the
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 Pi at every application a tail factor TF; = AET; × WCET; / 2. The tail factor has been randomly generated between 1 and 10. Thus, we have calculated the worst-case execution times as WCET; = TF; × AET; × 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 Pi at every application between 1 and 30 per cent of Pi’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 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 static scheduling algorithm, which is better than FTSS 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 $\mu$, $\zeta$, and $\rho$ 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 scheduleable and fault-tolerant solutions with the high overall utility.

8. References


