Scheduling of Fault-Tolerant Embedded Systems with Soft and Hard Timing Constraints

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

[1] {viaiz | petel | zebpe}@ida.liu.se
Dept. of Computer and Information Science
Linköping University

[2] Paul.Pop@imm.dtu.dk
Dept. of Informatics and Mathematical Modelling
Technical University of Denmark

Abstract

In this paper we present an approach to the synthesis of fault-tolerant schedules for embedded applications with soft and hard real-time constraints. We are interested to guarantee the deadlines for the hard processes even in the case of faults, while maximizing the overall utility. We use time/utility functions to capture the utility of soft processes. Process re-execution is employed to recover from multiple faults. A single static schedule computed off-line is not fault tolerant and is pessimistic in terms of utility, while a purely online approach, which computes a new schedule every time a process fails or completes, incurs an unacceptable overhead. Thus, we use a quasi-static scheduling strategy, where a set of 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. The proposed schedule synthesis heuristics have been evaluated using extensive experiments.

1. Introduction

Fault-tolerant embedded real-time systems have to be designed such that they implement correctly the required functionality even in the presence of 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 [8].

In addition, very important for the correct functioning of such systems are their timing constraints: "the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced" [10]. 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 [10]. However, many applications have both hard and soft timing constraints [2], and therefore researchers have proposed techniques for addressing mixed hard/soft real-time systems [2, 4, 3]. The issue of fault-tolerance has also been addressed separately for hard and soft systems.

In the context of hard real-time systems, researchers have shown that schedulability of hard real-time applications can be guaranteed for pre-emptive online scheduling [6, 11, 16]. However, such approaches lack the predictability required in many safety-critical applications, where static off-line scheduling is the only 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 [13]. The actual static schedules are generated according to the approach in [5]. Xie et al. [15] propose a technique to decide how replicas can be selectively inserted into the application, based on process criticality. Kandasamy et al. [9] propose constructive mapping and scheduling algorithms for transparent re-execution on multiprocessor systems. In [7] 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 [12]. In [1] 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 [14] 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.

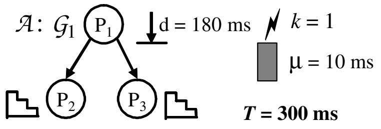
In this paper, we consider embedded systems composed of both hard and soft processes. Process re-execution is used to provide the required level of fault tolerance. We propose a novel quasi-static scheduling strategy, where a set of fault-tolerant schedules is synthesized off-line and, at run time, the 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 scheduling strategy can also handle overload situations with dropping of soft processes. The online overhead of quasi-static scheduling is very low, compared to traditional online scheduling approaches [3].

The next section presents our application model. Section 3 illustrates static and quasi-static scheduling for fault tolerance with examples. Section 4 presents our problem formulation. Static and quasi-static scheduling heuristics are discussed in detail in Section 5. Our experimental results, including a real-life example, are presented in Section 6.

2. Application Model

We model an application $\mathcal{A}$ as a directed, acyclic, polar graph $\mathcal{G}_k(\mathcal{V}_k, \mathcal{E}_k) \in \mathcal{A}$. 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 have arrived and it issues its outputs when it terminates. Processes are non-preemptable and thus cannot be interrupted 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 ex-



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

Figure 1. Application Example

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

978-3-9810801-3-1/DATE08 © 2008 EDAA

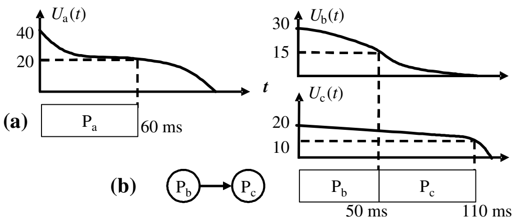


Figure 2. Soft Processes and Utility Functions

ecution time (BCET), t_i^b, an average-case execution time (AET), t_i^e, and a worst-case execution time (WCET), t_i^w. 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.

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_2 and P_3 are soft, while process P_1 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, with 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.

We consider that once a process has started, it completes until the end if no fault occurs. However, for a soft process P_i we have the option not to start it at all, and we say that we "drop" P_i, and thus its utility will be 0. This might be necessary in order to meet deadlines of hard processes, or to increase the overall system utility (e.g. by allowing other, more useful soft processes to complete). Moreover, if P_i is 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 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 of the formula to account for P_i itself. The intuition behind this formula is that the impact of stale value on P_i is in inverse proportion to the number of its inputs.

Suppose that soft process P_3 has two predecessors, P_1 and P_2. If P_1 is dropped 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 fault tolerance 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 a fault-scenario consisting of 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, and is detected by the error detection mechanism. After a worst-case recovery overhead of μ = 5 ms, depicted with a light gray rectangle, P_1 will be executed again. Its second execution in the worst-case could also experience a fault. Finally, the third execution 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, i.e., they can be dropped. 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. Static vs. Quasi-Static Scheduling

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 [7], that 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 $(t_i^w + \mu) \times f$, where f is the number of faults to tolerate. 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 as an f-schedule.

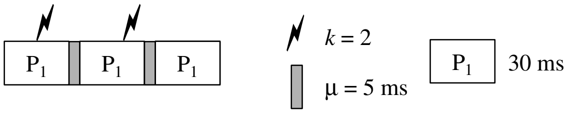


Figure 3. Re-execution

Let us illustrate how static scheduling would work for application $\mathcal{A}$ in Fig. 1. The application has to tolerate $k = 1$ faults and the recovery overhead μ is 10 ms for all processes. There are two possible ordering of processes: schedule S_1, "P_1, P_2, P_3" and schedule S_2, "P_1, P_3, P_2", for which the executions in the average case are shown in Fig. 4b$_1$-b$_2$. With a recovery slack of 70 ms, P_1 would meet the deadline in both of them and both schedules would complete before the period $T = 300$ ms. With a static scheduling approach we have to decide off-line, which schedule to use. In the average case for S_1, process P_2 completes at 100 ms and process P_3 completes at 160 ms. The overall utility in the average case is $U = U_2(100) + U_3(160) = 20 + 10 = 30$. In the average case for S_2, process P_3 completes at 110 and P_2 completes at 160, which results in the overall utility $U = U_3(110) + U_2(160) = 40 + 20 = 60$. Thus, S_2 is better than S_1 on average and is, hence, preferred. However, if P_1 will finish sooner, as shown in Fig. 4b$_5$, the ordering of S_1 is preferable, since it leads to a utility of $U = U_2(80) + U_3(140) = 40 + 30 = 70$, while the utility of S_2 would be only 60.

Hard processes have to be always executed and have to tolerate all k faults. Since soft processes can be dropped, this means that we do not have to re-execute them after a fault if their re-execution affects the deadline for hard processes, leads to exceeding the period T, or if their re-execution reduces the overall utility. In Fig. 4b$_4$, execution of process P_2 in the worst-case cannot complete within period T. Hence, process P_3 should not be re-executed. Moreover, in this example, dropping of $P_{3/2}$ is better for utility. If P_2 is executed instead of $P_{3/2}$, we get a utility of 10 even in the worst-case and may get utility of 20 if the execution of P_2 takes less time, while re-execution $P_{3/2}$ would lead to 0 utility.

In Fig. 4c, we reduce the period T to 250 for illustrative purposes. In the worst case, if process P_1 is affected by a fault and all processes are executed with their worst-case execution times, as shown in Fig. 4c$_1$, schedule S_2 will not complete within T. Neither will schedule S_1 do in Fig. 4c$_2$. Since hard process P_1 has to be fault-tolerant, the only option is to drop one of soft processes, either P_2 or P_3. The resulting schedules S_3: "P_1, P_3" and S_4: "P_1, P_2" are depicted in Fig. 4c$_3$ and Fig. 4c$_4$, respectively. The utility of S_3, $U{=}U_3(100) = 40$, is higher than the utility of S_4, $U{=}U_2(100) = 20$. Hence, S_3 will be chosen.

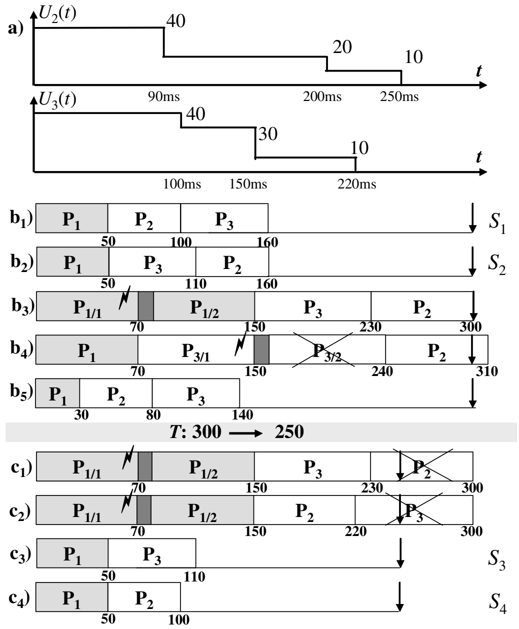


Figure 4. Static Scheduling

We have extended our approach from [7] to consider the hard/soft shared slacks, dropping of soft processes, and utility maximization for average execution times as presented in Section 5.2.

The problem with static scheduling is that there exists only one precalculated schedule and the application cannot adapt to a particular situation. In this paper we propose a quasi-static scheduling for fault tolerance to overcome the limitations of static scheduling. The main idea of quasi-static scheduling is to generate off-line a set of schedules, each adapted to 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 guarantees the hard deadlines and maximizes utility) depending on the occurrence of faults and the actual execution times of processes.

The set of schedules is organized as a tree, where each node corresponds to a schedule, and each arc is a schedule switch that has to be performed if the condition on the arc becomes true during the execution. Let us illustrate such a tree in Fig. 5, for the application $\mathcal{A}$ in Fig. 1. We will use utility functions depicted in Fig. 4a. The quasi-static tree is constructed for the case $k = 1$ and contains 12 nodes. We group the nodes into 4 groups. Each schedule is denoted with S_i^j, where j stands for the group number. Group 1 corresponds to the no-fault scenario. Groups 2, 3 and 4 correspond to a set of schedules in case of faults affecting processes P_1, P_2, and P_3, respectively. The schedules for the group 1 are presented in Fig. 5b. The scheduler starts with the schedule $S_1^{\,1}$. If process P_1 completes after 40, the scheduler switches to schedule $S_2^{\,1}$, which will produce a higher utility. If the fault happens in process P_1, the scheduler will switch to schedule $S_1^{\,2}$ that contains the re-execution $P_{1/2}$ of process P_1. Here we switch not because of utility, but because of fault tolerance. Schedules for group 2 are depicted in Fig. 5c. If the re-execution $P_{1/2}$ completes between 90 and 100, the scheduler switches from $S_1^{\,2}$ to $S_2^{\,2}$, that gives higher utility, and, if the re-execution completes after 100, it switches to $S_3^{\,2}$ in order to satisfy timing constraints. Schedule $S_3^{\,2}$ represents the situation illustrated in Fig. 4c$_2$, where process P_3 had to be dropped. Otherwise, execution of process P_3 will exceed the period T. Note that we choose to drop P_3, not P_2, because this gives a higher utility value.

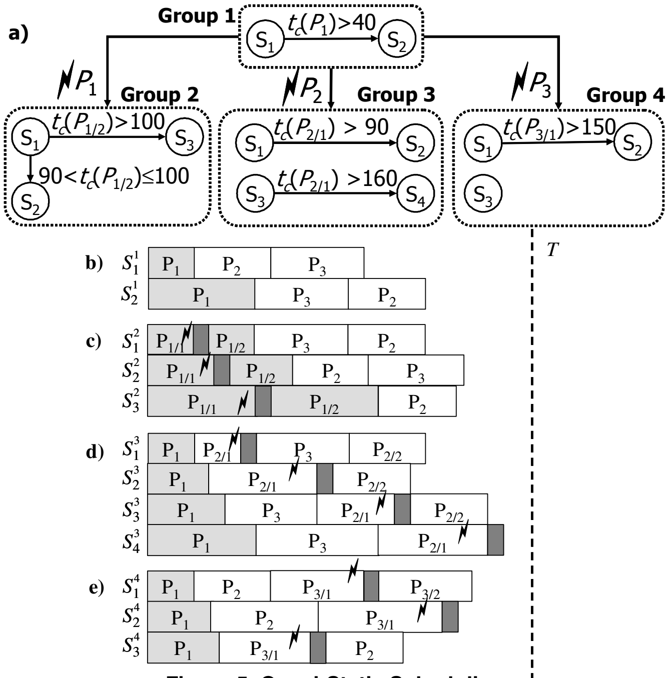


Figure 5. Quasi-Static Scheduling

The generation of a complete quasi-static tree with all the necessary schedules that captures different completion times of processes is practically infeasible for large applications. The number of fault scenarios is growing exponentially with the number of faults and the number of processes [8]. In addition, in each fault scenario, the processes may complete at different time moments. The combination of different completion times is also growing exponentially [3]. Thus, the main challenge of quasi-static scheduling is to generate an as small as possible number of schedules with the most improvement to the overall utility.

4. Problem Formulation

As an input we get an application $\mathcal{A}$, represented as an acyclic directed polar graph $\mathcal{G}$, with a set $\mathcal{S}$ of soft processes and set $\mathcal{H}$ 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. We also know the best, average, and worst-case execution times for each process, as presented in Section 2.

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 scenario, maximizes the overall utility U^f in faulty scenarios, and satisfies all hard deadlines in all scenarios. 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. This property will be captured in all our algorithms.

5. Scheduling Strategy

Due to complexity, in our approach we restrict the number of schedules that are part of the quasi-static tree. Our quasi-static scheduling strategy for fault tolerance is presented in Fig. 6. We are interested in determining the best M schedules that will guarantee the hard deadlines (even in the case of faults) and maximize the overall utility. Thus, the function returns either a fault-tolerant quasi-static tree Φ of size M or that the application is not schedulable.

We start by generating the *f*-schedule S_{root}, using the static scheduling algorithm for fault tolerance (FTSS) presented in Section 5.2, which considers the situation where all the processes are executed with their worst-case execution times, while the utility is maximized for the case where processes are executed with their average execution times (as was discussed in Fig. 4). Thus, 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 the online scheduler to re-execute processes online, without changing the order of process execution. Since this is the schedule assuming the worst-case execution times, many soft processes will be dropped to provide a schedulable solution.

If the *f*-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 *f*-schedule S_{root} is schedulable, we generate the quasi-static tree Φ starting from schedule S_{root} by calling the FTQS heuristic presented in Section 5.1, which uses FTSS to generate *f*-schedules that maximize utility.

```
SchedulingStrategy($\mathcal{G}$, $k$, $M$)
  $S_{root}$ = FTSS($\mathcal{G}$, $k$)
  if $S_{root} = \varnothing$ then return unschedulable
  else
     set $S_{root}$ as the root of fault-tolerant quasi-static tree $\Phi$
     $\Phi$ = FTQS($\Phi$, $S_{root}$, $k$, $M$)
     return $\Phi$
  end if
end SchedulingStrategy
```

Figure 6. General Scheduling Strategy

5.1 Quasi-Static Scheduling

In general, quasi-static scheduling should generate a tree that will adapt to different execution situations. However, tracing all execution scenarios is infeasible. Therefore, we have used the same principle as in [3] to reduce the number of schedules in the quasi-static tree Φ, where only best-case and the worst-case execution times of processes are considered.

Our quasi-static scheduling for fault tolerance (FTQS) heuristic, outlined in Fig. 7, generates a fault tolerant quasi-static tree Φ of a given size M for a given root schedule S_{root}, which tolerates k faults. Schedule S_{root} is generated such that each process P_i completes within its worst-case execution time (see our strategy in Fig. 6), including soft processes.

At first, we explore the combinations of best- and worst-case execution times of processes by creating sub-schedules from the root schedule S_{root} (line 2). We generate a sub-schedule SS_i for each process P_i in S_{root}. SS_i begins with process P_i executed with its best-case execution time. The rest of the processes in SS_i, after P_i, are scheduled with the FTSS heuristic, which generates an *f*-schedule for the worst-case execution times, while the utility is maximized for average execution times.

After producing the *first layer* of sub-schedules (from root schedule S_{root}), a *second layer* of sub-schedules is created. For each sub-schedule SS_i on the first layer, which begins with process P_i, we create with FTSS the second-layer sub-schedule SS_j for each process P_j after process P_i. Each initial process P_j of the sub-schedule SS_j is executed with its best-case execution time. Similarly, we generate the sub-schedules of the third layer. Unless we terminate the heuristic, the generation of sub-schedule layers will continue until all combinations of best- and worst-case execution times of processes are reflected in the tree Φ.

Although, in principle, all the sub-schedules can be captured in the quasi-static tree Φ, it would require a lot of memory because the number of sub-schedules is growing exponentially with the number of processes in the application. Therefore, we have to keep only those sub-schedules in the tree that, if switched to, lead to the most significant improvement in terms of the overall utility. In general, our strategy is to eventually generate the most different sub-schedules. We limit the tree size to M and, when the number of different schedules in the tree Φ reaches M, we stop the exploration (line 3).

Our fault-tolerant quasi-static tree Φ finally contains schedules generated for only the best-case and worst-case execution times of processes. However, the actual execution times of processes will be somewhere between the best-case and the worst-case. Therefore, in the quasi-static tree we have to provide information when it is better to switch from "parent" schedule SS_P to a sub-schedule SS_i after process P_i is completed. The completion times of process P_i may vary from the best-possible, when all processes scheduled before P_i and P_i itself are executed with their best-case execution

```
FTQS($\Phi$, $S_{root}$, $k$, $M$)
 1 layer = 1
 2 $\Phi$ = $\Phi \cup$ CreateSubschedules($S_{root}$, k, layer)
 3 while DifferentSchedules($\Phi$) < M do
 4    $SS_p$ = FindMostSimilarSubschedule($\Phi$, layer)
 5    if $SS_p = \varnothing$ then return layer = layer + 1
 6    else
 7       $\Phi$ = $\Phi \cup$ CreateSubschedules($SS_p$, k, layer + 1)
 8    end if
 9 end while
10 IntervalPartitioning($\Phi$)
11 return $\Phi$
end FTQS
```

Figure 7. Quasi-Static Scheduling Algorithm

times, to the worst-possible, which is the worst-case fault scenario (with k faults) when all processes before P_i and P_i itself are executed with the worst-case execution times. We trace all possible completion times of process P_i , assuming they are integers, and compare utility values produced by SS_P and SS_i (line 10). This procedure is called *interval-partitioning* [3]. If the utility value produced by SS_i is greater than the utility value produced by SS_P , then switching to schedule SS_i makes sense. SS_i is not always safe since it considers best-case execution time of P_i. SS_i will violate deadlines after certain completion time t_i^c. Therefore, if P_i completes after t_i^c , then SS_P schedule has to be used.

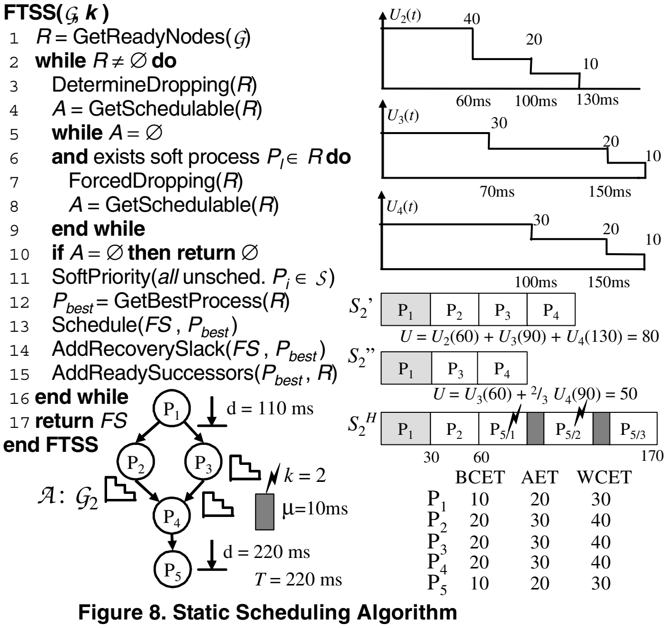
After interval partitioning is done, FTQS returns a fault-tolerant quasi-static tree Φ, which can be used by the online scheduler.

5.2 Static Scheduling for Fault Tolerance

Our static scheduling for fault tolerance and utility maximization (FTSS), 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 have been scheduled. The heuristic initializes the ready list R with processes ready at the beginning (line 1) and is looping while there is at least one process in the list (line 2).

FTSS addresses the problem of dropping of soft processes. All soft processes in the ready list R are evaluated if they can be dropped (line 3). To determine exactly whether a particular soft process P_i should be dropped, we have to generate two schedules with and without process P_i. However, in each of these schedules other processes have to be also evaluated for dropping, and so on. Instead of evaluating all possible dropping combinations, we use the following heuristic: for each process P_i we generate two schedules, S_i' and S_i", which contain only unscheduled soft processes. Schedule S_i' contains P_i, while schedule S_i" does not. In schedule S_i", if $U(S_i') \le U(S_i")$, P_i is dropped and the stale value is passed instead. In Fig. 8 we depict S_2' and S_2" for process P_2 in application $\mathcal{A}$ (presented in the bottom of Fig. 8). We check if we can drop P_2. S_2', which contains P_2, produces a utility of 80, while S_2" produces a utility of only 50. Hence, process P_2 will not be dropped. If a soft process is dropped, its "ready" successors are put into the ready list.

After removing soft processes from the ready list R, we select a set A of processes from R that would lead to a schedulable solution (even in case of k faults), line 4. For each process $P_i \in R$ the schedule S_i^H, which contains process P_i and unscheduled hard processes, is generated. This schedule is the shortest valid schedule containing process P_i, where all (other) soft processes have been dropped. If the hard deadlines are met, then P_i leads to a schedulable solution. In Fig. 8 we have presented schedule S_2^H for application $\mathcal{A}$. We evaluate if process P_2 is schedulable. The only unscheduled hard process P_5 completes at 170 ms in the worst-case fault scenario with two faults, which is before its deadline of 220 ms. Thus deadlines are met and P_2 is schedulable.



```
FTSS($\mathcal{G}$, k )
1  R = GetReadyNodes($\mathcal{G}$)
2  while R ≠ ∅ do
3     DetermineDropping(R)
4     A = GetSchedulable(R)
5     while A = ∅
6     and exists soft process $P_l \in$ R do
7        ForcedDropping(R)
8        A = GetSchedulable(R)
9     end while
10    if A = ∅ then return ∅
11    SoftPriority(all unsched. $P_l \in \mathcal{S}$)
12    $P_{best}$ = GetBestProcess(R)
13    Schedule(FS , $P_{best}$)
14    AddRecoverySlack(FS , $P_{best}$)
15    AddReadySuccessors($P_{best}$ , R)
16 end while
17 return FS
end FTSS
```

	BCET	AET	WCET
P_1	10	20	30
P_2	20	30	40
P_3	20	30	40
P_4	20	30	40
P_5	10	20	30

Figure 8. Static Scheduling Algorithm

If none of the processes in ready list R is leading to a schedulable solution, one of the soft processes is removed from the ready list and its successors are put there instead. We choose that soft process which, if dropped, would reduce the overall utility as little as possible (lines 5–9). Then, the set A is recalculated. If no schedulable process is found, the application is not schedulable and the algorithm returns ∅ (line 10).

The next step is to find which process out of the schedulable processes is the best to schedule. We calculate priorities for *all* unscheduled soft processes using the MU function presented in [3] (line 11). The GetBestProcess function (line 12) selects either best soft process P_s with highest priority SP_s or, if there are no soft processes in the ready list, the hard process P_h with the earliest deadline.

Once process P_{best} is scheduled (line 13), the recovery slack with the number of re-execution has to be assigned to it (line 14). For the 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. If the re-execution is schedulable, it is evaluated with the dropping heuristic. If it is better to drop the re-execution, then we drop it.

After assigning the recovery slack for process P_{best} , we remove process P_{best} from the ready list and add P_{best}'s ready successors into it (lines 15).

FTSS returns an f-schedule FS generated for worst-case execution times, while the utility is maximized for average execution times of processes.

6. Experimental Results

For the experiments, we have generated 450 applications with 10, 15, 20, 25, 30, 35, 40, 45, and 50 processes, where we have uniformly varied worst-case execution times of processes between 10 and 100 ms. We have generated best-case execution times between 0 ms and the worst-case execution times. We consider that completion time of processes is uniformly distributed between the best-case execution time t_i^b and the worst-case execution time t_i^w, i.e. the average execution time t_i^e is $(t_i^w - t_i^b)$ / 2. The number k of tolerated faults has been set to 3 and the recovery overhead μ to 15 ms. The experiments have been run on a Pentium 4 2.8 GHz processor with 1Gb of memory.

In the first set of experiments we have evaluated the quality of the static fault-tolerant schedules produced by our FTSS algorithm. We have compared with a straightforward approach that works as follows: we obtain static non-fault-tolerant schedules that produce maximal value (e.g. as in [3]). Those schedules are then made fault-tolerant by adding recovery slacks to tolerate k faults in hard processes. The soft processes with lowest utility value are dropped until the application becomes schedulable. We call this straightforward algorithm FTSF. We can see in Fig. 9 that FTSF is 20-70% worse in terms of utility compared to FTSS.

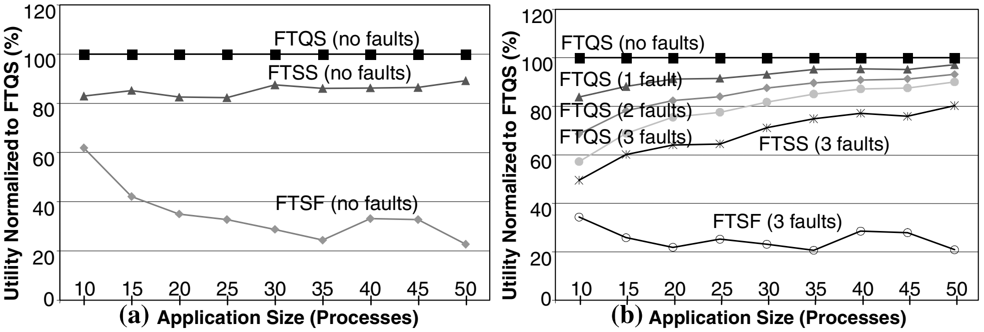


Figure 9. Comparison between FTQS, FTSS, and FTSF

Table 1. Increasing the Number of Nodes for FTQS

Nodes	Utility Normalized to FTSS (%)				Run time, sec
	0	**1**	**2**	**3**	
1	**100**	**93**	**88**	**82**	**0.62**
2	**111**	104	97	91	1.17
8	**121**	113	106	99	2.48
13	**122**	114	107	100	3.57
23	**124**	115	107	100	4.78
34	**125**	117	109	102	8.06
79	**125**	117	110	102	26.14
89	**126**	117	110	102	38.79

In a second set of experiments we were interested to determine the quality of our quasi-static approach for fault tolerance (FTQS) in terms of overall utility for the no-fault scenario and for fault scenarios. Fig. 9 presents the normalized utility obtained by the three approaches, varying the size of applications. We have evaluated schedules generated by FTQS, FTSS, and FTSF with extensive simulations. We considered 20,000 different execution scenarios for the case of no faults, 1, 2, and 3 faults (in order not to overload the figure, for FTSS and FTSF only the 3 faults case is depicted). The overall utility for each case is calculated as an average over all execution scenarios. Fig. 9a shows the results for the no-fault scenarios. We can see that FTQS is 11-18% better than FTSS which is the best of the static alternatives.

We were also interested how FTQS performs for the cases when faults happen. Fig. 9b shows the normalized utility in case of faults. Obviously, as soon as a fault happens, the overall produced utility is reduced. Thus, in case of a single fault, the utility of schedules produced with FTQS goes down by 16% for 10 processes and 3% for 50 processes. The utility is further reduced if 2 or all 3 faults are occurring: with 31% and 43% for 10 processes and with 7% and 10% for 50 processes, respectively. FTQS is constantly better than the static alternatives which demonstrates the importance of dynamically taking decisions and being able to chose among efficient precalculated scheduling alternatives.

In the third set of experiments, we were interested to evaluate the quality of FTQS in terms of the quasi-static tree size. Less nodes in the tree means that less memory is needed to store them. Therefore, we would like to get the best possible improvement with fewer nodes. We have chosen 50 applications with 30 processes each and set the percentage of soft and hard processes as 50/50 (i.e. half of each type). The results are presented in Table 1, for 0, 1, 2, and 3 faults, where, as a baseline, we have chosen FTSS, which generates a single *f*-schedule. As the number of nodes in the tree is growing, the utility value is increasing. For example, with two nodes it is already 11% better than FTSS and with 8 nodes it is 21% better. Finally, we reach 26% improvement over FTSS with 89 nodes in the tree. The runtime of FTQS also increases with the size of the quasi-static tree (from 0.62 sec for FTSS to 38.79 sec for FTQS with 89 nodes).

We have also run our experiments on a real-life example, a vehicle cruise controller (CC) composed of 32 processes [8], which is implemented on a single microcontroller with a memory unit and communication interface. Nine processes, which are critically involved with the actuators, have been considered hard. We have set $k = 2$ and have considered μ as 10% of process worst-case execution times. FTQS requires 39 schedules to get 14% improvement over FTSS and 81% improvement over FTSF in case of no faults. The utility of schedules produced with FTQS is reduced by 4% with 1 fault and by only 9% with 2 faults.

7. Conclusions

In this paper we have addressed fault-tolerant applications with soft and hard real-time constraints. The timing constraints were captured using deadlines for hard processes and time/utility functions for soft processes.

We have proposed an approach to the synthesis of fault-tolerant schedules for fault-tolerant mixed hard/soft applications. Our quasi-static scheduling approach guarantees the deadlines for the hard processes even in the case of faults, while maximizing the overall utility of the system.

The experiments have shown that our approach selects online the right precalculated schedules in order to meet the timing constraints and deliver high utility even in case of faults.

References

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

[2] 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.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

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

[9] 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.

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

[11]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.

[12]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.

[13]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.

[14]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.

[15]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.

[16]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.