Value-Based Scheduling of Distributed Fault-Tolerant Real-Time Systems with Soft and Hard Timing Constraints

Viacheslav Izosimov

Embedded Intelligent Solutions (EIS) By Semcon AB
Box 407, SE-581 04 Linköping, Sweden
E-mail: viacheslav.izosimov@eis.semcon.com

Petru Eles, Zebo Peng
Dept. of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden
E-mail: {petru.eles | zebo.peng}@liu.se

Abstract - **We present an approach for scheduling of fault-tolerant embedded applications composed of soft and hard real-time processes running on distributed embedded systems. The hard processes are critical and must always complete on time. A soft process can complete after its deadline and its completion time is associated with a value function that characterizes its contribution to the quality-of-service of the application. We propose a quasi-static scheduling algorithm to generate a tree of fault-tolerant distributed schedules that maximize the application's quality value and guarantee hard deadlines.**

Keywords - mixed soft and hard real-time, fault tolerance, distributed embedded systems, quasi-static scheduling, utility maximization, quality-of-service

I. Introduction

Modern embedded applications, used, for example, in multimedia devices and automotive infotainment, are often executed on distributed systems. They should deliver highest-possible quality-of-service and meet timing constraints even in the presence of faults. We model quality-of-service and safety constraints, by separation of concerns, as soft and hard real-time processes [3]. A soft process can complete after its deadline and its completion time is associated with a value function that characterizes its contribution to the quality-of-service of the application. A soft process can be dropped to let hard processes or more important soft processes execute instead. Hard processes represent time-constrained parts of the application, which must be always executed and meet deadlines.

Static off-line scheduling is an attractive option for embedded applications since it can ensure both the predictability of worst-case behavior and high resource utilization [15]. However, off-line scheduling lacks flexibility and, unless extended with adaptive functionality, cannot handle overloads or provide efficient fault recovery [14, 6]. Traditional off-line worst-case-driven designs are also overly pessimistic because the worst case often corresponds to rare execution scenarios, which is especially critical for multimedia applications [3]. In this paper, we overcome the limitations of traditional off-line scheduling by employing a quasi-static scheduling technique.

As a part of our design, we deal with transient and intermittent faults[1] (also known as "soft errors"), which belong to the most common types of system faults today. Such faults appear for a short time without causing permanent damage, and can be caused by electromagnetic interference, radiation, temperature variations, software "bugs", etc. [10]. Online scheduling with fault tolerance constraints for hard real-time systems has been considered in [8, 16, 23]. Researchers have also proposed a number of approaches for integrating fault tolerance into the framework of static scheduling for hard real-time systems [18, 22, 14, 9, 20]. Xie et al. [22] have proposed a technique to decide how replicas can be selectively inserted into the application, based on process criticality. Pinello et al. [18] have proposed a simple heuristic for combining several static schedules in order to mask fault patterns against primarily permanent faults. Kandasamy et al. [14] have proposed constructive mapping and scheduling algorithms for transparent re-execution on multiprocessor systems but only considered one fault per computation node. In [9, 20] we have proposed scheduling and fault tolerance policy assignment techniques for distributed real-time systems, such that the required level of fault tolerance is achieved and real-time constraints are satisfied with a limited amount of resources.

Regarding soft real-time systems, the available approaches in [1, 17, 21] address fault tolerance only in the context of online scheduling. In [17] researchers have shown how faults can be tolerated with active replication while maximizing the quality level of the system. An online greedy resource allocation algorithm has been proposed, which incrementally chooses waiting process replicas and allocate them to the least loaded processors. 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. The approach only considers monoprocessor architectures. In [21] the 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 the results. None of the above approaches considers value-based scheduling optimization in the context of static cyclic scheduling. In general, the considered value-based optimization is either very limited and based on costly active

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

replication [17, 21] or restricted to monoprocessor systems with online scheduling [1].

None of the above approaches considers systems composed of both soft and hard real-time processes. Hard and soft real-time systems have been traditionally scheduled using very different techniques [15]. However, many embedded applications have both components with hard and soft timing constraints [3]. Therefore, several techniques for scheduling of mixed hard/soft real-time systems have been proposed [3, 5, 4]. However, none of this work addresses fault tolerance aspects. Thus, in [11], we have used process re-execution together with a quasi-static scheduling strategy to provide fault tolerance in the context of mixed soft and hard real-time systems. We have proposed a quasi-static scheduling strategy, which could handle dropping of soft processes. In [12], we have enhanced our monoprocessor strategy with preemption. However, our approaches in [11, 12] are restricted to monoprocessor systems and cannot be utilized for distributed embedded systems.

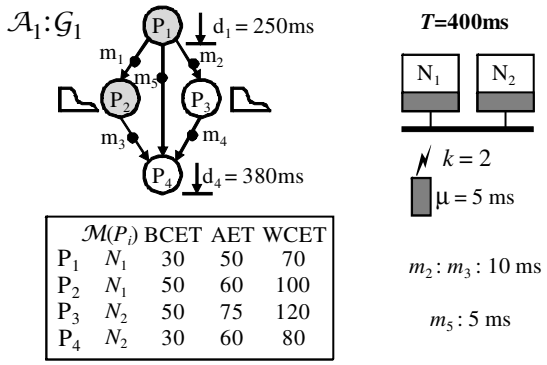
In this work, we propose an approach for scheduling of fault-tolerant applications composed of soft and hard real-time processes running on distributed systems. We use process re-execution to tolerate transient faults. We propose a quasi-static scheduling algorithm that generates off-line a tree of fault-tolerant schedules that maximize the quality-of-service of the application and, at the same time, guarantee deadlines for hard processes. At run time, the online scheduler, with very low online overhead, would select the appropriate schedule based on the occurrence of faults and the actual execution times of processes.

The approach presented in this paper is different from our previous approaches [11, 12] because, first of all, we propose a value-based scheduling for multiprocessor systems, which is able to generate schedules to maximize the overall utility in the distributed system context. We have also developed a signalling mechanism to transmit knowledge of the global system state from one computation node to the others and we explicitly account for communications on the bus during generation of schedule tables. Our approach can be useful on any distributed system whose worst-case communication delays can be obtained. Examples of such systems include, but are not limited to, multimedia systems in mobile phones, media players, TV-sets, and automotive infotainment. Our approach is also applicable for safety-critical systems used, for example, in factory automation or automobiles.

The next section presents our application model, fault tolerance and our utility value model. In Section III, we outline our problem formulation. Section IV presents a motivational example to illustrate scheduling decisions. In Section V, we present our quasi-static scheduling heuristics. Experimental results are presented in Section VI.

II. Application Model

We model an application $\mathcal{A}$ as a set of directed, acyclic graphs merged into a single hypergraph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. Each node $P_i \in \mathcal{V}$ represents one process. An edge $e_{ij} \in \mathcal{E}$ from P_i to P_j indicates that the output of P_i is the input of P_j. A process P_k or the communication captured by an edge e_{ij} can be either mandatory (*hard*) or optional (*soft*). A process can be activated after all its hard inputs, required for the execution, have arrived. A process issues its outputs when it terminates. Process executions are not pre-emptable.



	$\mathcal{M}(P_i)$	BCET	AET	WCET
P_1	N_1	30	50	70
P_2	N_1	50	60	100
P_3	N_2	50	75	120
P_4	N_2	30	60	80

Figure 1. Application Example

We consider that the application is running on a set of heterogeneous computation nodes $\mathcal{N}$ connected to a bus B. Mapping of processes in the application to the computation nodes is determined by a function $\mathcal{M}$: $\mathcal{V} \rightarrow \mathcal{N}$. For a process $P_i \in \mathcal{V}$, $\mathcal{M}(P_i)$ is the node to which P_i is assigned for execution. We know for each process P_i a best-case execution time (BCET), t_i^b, and a worst-case execution time (WCET), t_i^w, on P_i's computation node $\mathcal{M}(P_i)$. The execution time distribution $E_i(t)$ of P_i on node $\mathcal{M}(P_i)$ is given. An expected execution time (AET) for P_i on node $\mathcal{M}(P_i)$, t_i^e, is obtained from the execution time distribution $E_i(t)$.

Processes mapped on different computation nodes communicate by messages sent over the bus. We consider that the worst-case sizes of messages are given, which can be implicitly translated into the worst-case transmission times on the bus. If processes are mapped on the same node, the message transmission time between them is accounted for in the worst-case execution time of the sending process.

In Fig. 1 we have an application $\mathcal{A}_1$ consisting of the process graph $\mathcal{G}_1$ with four processes, P_1 to P_4. Processes P_1 and P_2 are mapped on computation node N_1, and P_3 and P_4 on N_2. The execution times for processes and transmission times of messages are shown in the table. Processes P_1 and P_4 and message m_5 are hard, while processes P_2 and P_3 and messages m_1, m_2, m_3, m_4 are soft. Hard processes are associated with *deadlines*, while soft processes are associated with *utility functions* as discussed in Section II.B.

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. Process graphs with different periods are combined into a hyper-graph, capturing all process activations for the hyper-period (LCM of all periods). In Fig. 1 process graph $\mathcal{G}$ has a period $T = 400$ ms.

A. Fault Tolerance

In our model, we consider that at most k transient faults may occur during a hyper-period T of the application. The number f of faults in a particular execution scenario can be less or equal

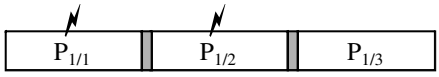


Figure 2. Re-execution

to the maximum number k. We consider that a fault scenario with $f-1$ (less) faults is more likely than a fault scenario with f (more) faults and, thus, the no fault scenario is the most likely to happen. We use re-execution for tolerating faults [14, 9, 20]. Process re-execution requires an additional time overhead μ. For example, for application $\mathcal{A}_1$ in Fig. 1, $k = 2$ and $\mu = 5$ ms.

The software architecture, including the real-time kernel, error detection and fault-tolerance mechanisms are themselves fault-tolerant, i.e., they employ extensive internal fault tolerance mechanisms. We assume that transient faults on the bus are addressed at the communication level, for example, with the use of efficient error correction codes [19, 2] and/or acknowledgements/retransmissions [13].

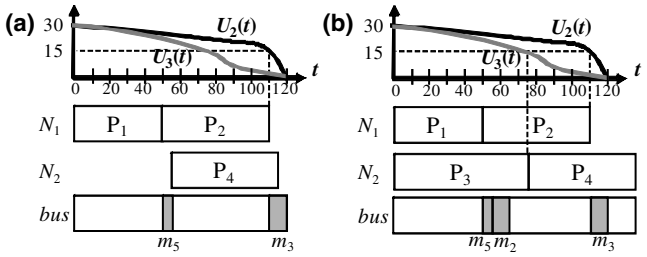
In this paper, we will denote with $P_{i/j}$ the jth execution of process P_i in the faulty scenario, where P_i is affected by faults. Let us consider the example in Fig. 2, where we have $k = 2$. In the worst case, as depicted in Fig. 2, the first fault happens during P_1's first execution, denoted $P_{1/1}$. The error detection overhead is considered as part of the process execution time. After a worst-case recovery overhead of $\mu = 5$ ms, depicted with a light gray rectangle, P_1 is executed again. Its second execution $P_{1/2}$ in the worst-case can also experience a fault. Finally, the third execution $P_{1/3}$ of P_1 succeeds.

B. Utility Model

Each hard process $P_i \in \mathcal{V}$ is associated with a hard deadline d_i. In Fig. 1, process P_1 has deadline $d_1 = 250$ ms and process P_4 has deadline $d_4 = 380$ ms. Although hard messages are not associated with deadlines, they provide critical communications and must be always transmitted. Each soft process $P_i \in \mathcal{V}$ is assigned with a utility function $U_i(t)$, which can be any non-increasing monotonic function of the completion time of a process. The *overall utility* of each execution scenario of the application is the sum of individual utilities produced by soft processes in this scenario.[1]

In Fig. 3 we depict utility function $U_2(t)$ for the soft process P_2 of application $\mathcal{A}_1$ in Fig. 1. According to the schedule in Fig. 3a, P_2 completes at 110 ms and its utility would be 15. For a soft process P_i we have the option to "drop" the process, and, thus, its utility will be 0, i.e., $U_i(-) = 0$. In Fig. 3a we drop process P_3. Thus, the overall utility of the application in this case will be $U = U_2(110) + U_3(-) = 15$. We may also drop soft messages of the application alone or together with the producer process. For example, in Fig. 3a, message m_2 is dropped. 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).[2]

1. Note that in this work we do not assign utility functions to hard real-time processes in order to clearly separate hard and soft real-time properties of the application, which, in turn, allows us to efficiently meet these properties in the design optimization and implementation phases.



$U = U_2(110) + U_3(-) = 15 + 0 =$ **15** $U=U_2(110)+U_3^*(75)=15+{}^1/_2\times15=$**22.5**

Figure 3. Utility Functions and Dropping

If P_i, or its re-execution, is dropped but would 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. Thus, output values produced by processes in one execution cycle can be reused by processes in the next cycles. Reusing stale inputs, however, may lead to reduction in the utility value, i.e., utility of a process P_i would degrade 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 and messages. α_i is obtained according to an application-specific rule $\mathcal{R}$, given by designers, which specifies service degradation properties for a particular application. In this work, we will consider that if a soft process P_i or its re-executions are dropped, then $\alpha_i = 0$, i.e., its utility $U_i^*(t)$ will be 0. If P_i completes, but uses stale inputs from one or more of its direct predecessors, the stale value coefficient is 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.[3]

If we apply the above rule to the execution scenario on the schedule in Fig. 3b, the overall utility is $U=U_2(110) + U_3^*(75) = 15 + {}^1/_2 \times 15 = 22.5$. The utility of process P_3 is degraded because it uses a "stale" value from process P_1, i.e., P_3 does not wait for input m_2 and its stale value coefficient $\alpha_3 = (1 + \alpha_1) / (1+|DP(P_3)|) = (1 + 0) / (1 + 1) = {}^1/_2$.[4]

Note that, although in this work we will use the service degradation rule presented above, this rule is *an example* of a

2. For example, an image processing automotive system for pedestrian detection will execute hard processes to perform the necessary functions to follow pedestrians in the most "dangerous" areas, but does not have to follow *all* the pedestrians *at every cycle*. The latter would increase the quality of pedestrian detection but may not be affordable, for example, in the presence of transient faults or system overload.

3. In this work we will consider that, if the soft process is dropped in one execution cycle, it will not degrade utility values of its predecessors since the outputs from the predecessors can be re-used in the next execution cycle (when, for example, a predecessor is dropped or the soft process is forced to be executed before its predecessor).

4. Note that soft process P_3 is executed before its hard predecessor P_4 because, otherwise, P_3 would deliver utility of 0 according to its utility function. This, in fact, is an example of a "bad" design, which we have presented primarily for illustrative purposes. Such designs, in general, should be avoided by designers unless there exist particular reasons for not doing so, for example, due to imposed system requirements, which cannot be overcome.

possible service degradation rule, which can be imposed by designers. Our approach can be used with any other service degradation rule, different from the one considered above.

III. Problem Formulation

As an input we get an application $\mathcal{A}$, as introduced in Section II. Application $\mathcal{A}$ runs with a period T on a bus-based architecture composed of a set $\mathcal{N}$ of heterogeneous computation nodes. The mapping of processes on the set $\mathcal{N}$of computation nodes is given. We know the best, worst and expected execution times for each process on the computation node on which this process is mapped, as presented in Section II. The maximum number k of transient faults and the recovery overhead μ are given.

As an output, we have to obtain a quasi-static tree of fault-tolerant schedules on computation nodes and the bus that

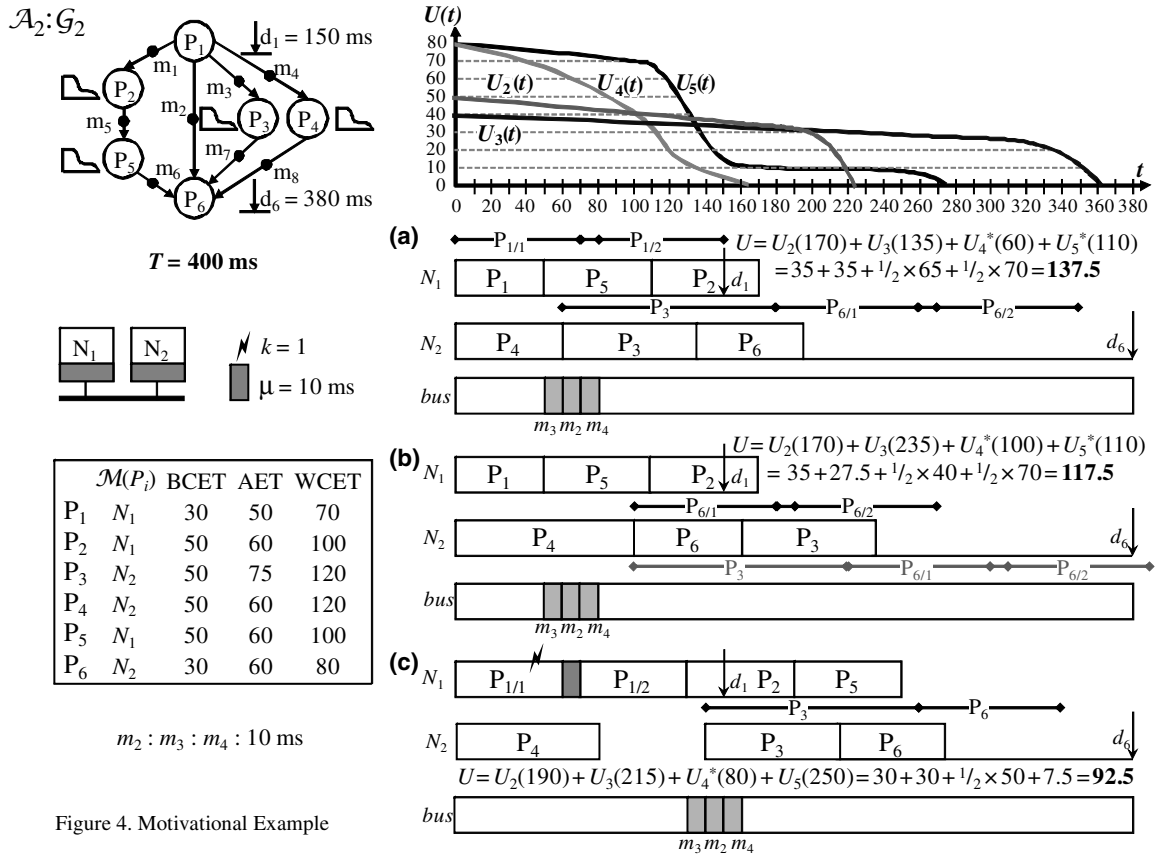
- firstly, maximizes the total utility U of the application in the no-fault scenarios,
- secondly, maximizes the total utility U^f in fault scenarios, and
- satisfies all hard deadlines in all scenarios.

Schedules must be generated under the assumption that the no-fault scenario is the most likely to happen, and scenarios with less faults are more likely than those with more faults. This means that schedules for $f + 1$ faults should not compromise schedules for f faults.

IV. Motivational Example

Application $\mathcal{A}_2$ in Fig. 4 is composed of 6 processes, P_1 to P_6, where processes P_2 to P_5 are soft with utility functions $U_2(t)$ to $U_5(t)$ and processes P_1 and P_6 are hard with deadlines 150 and 380 ms, respectively. Message m_2 is hard and all other messages are soft.[1] In Fig. 4a-c, we depict three schedules for three possible execution scenarios for application $\mathcal{A}_2$. In Fig. 4a, processes are executed with their expected execution times and we take a number of scheduling decisions to increase utility. Process P_4 does not wait for input from P_1, which gives an increased utility U_4 of $^1/_2 \times 65 = 32.5$, instead of only 20 (in fact, process P_4 will always get more utility, due to the shape of its utility function if it uses a "stale" value from P_1). P_5 is executed before its soft predecessor P_2 with an increased utility U_5 of $^1/_2 \times 70 = 35$, instead of only 10 as if it would follow P_2. Finally, P_2 and P_3 get their inputs from P_1 and complete with utility of 35 each. Thus, the overall utility is 137.5. The schedule is safe since processes P_1 and P_6 meet deadlines even in the worst-case fault scenarios. We depict the lengths of the worst-case scenarios above the Gantt charts of the schedules. On N_1, P_1's recovery meets deadline. The worst-case scenario

1. An example of the system implementing process graph $\mathcal{G}_2$ could be an on-board GPS (Global Positioning System), which (*i*) reads the driver's inputs, connects to the minimum number of satellites, loads a basic road map (all with process P_1) and (*ii*) returns the decision on the direction to the driver (with process P_6). These two processes are mandatory and have to be executed before the imposed hard deadlines. The decision on the direction based on only the output from process P_1 is not optimal and can be improved. GPS can connect to more satellites to increase the precision with process P_4. It can download an updated road map with process P_2 and adjust the decision based on this map with process P_5. It can also read information about the speed cameras and directly warn the driver with process P_3, while also influencing the final decision of process P_6.



	$\mathcal{M}(P_i)$	BCET	AET	WCET
P_1	N_1	30	50	70
P_2	N_1	50	60	100
P_3	N_2	50	75	120
P_4	N_2	50	60	120
P_5	N_1	50	60	100
P_6	N_2	30	60	80

$m_2 : m_3 : m_4$: 10 ms

Figure 4. Motivational Example

for P_6 contains the worst-case execution of P_3 and the re-execution of P_6, where re-execution of P_3 is dropped.

In the scenario of Fig. 4b process P_4 is executing for 100 ms. To meet the deadline of P_6, the execution of P_3 is postponed and P_6 is moved forward. The overall utility is, hence, degraded to 117.5. Above the schedule for N_2 we depict the worst-case situation, which contains only re-execution of P_6, while execution of P_3, in this case, is dropped. Under the schedule for N_2 we depict the worst-case scenario that would be produced if keeping the same order as in Fig. 4a. It can be observed that P_6 would miss its deadline. Thus, changing of the order, despite utility degradation, is necessary.

In Fig. 4c, we show a fault scenario, where process P_1 is re-executed with re-execution completed at 130 ms. The order in the schedule should be changed again. Process P_5 is postponed and executed after its predecessor P_2. It will lead to the increased utilities U_2 and U_5 of $30 + 7.5 = 37.5$, instead of $27.5 + {}^1/_2 \times 10 = 32.5$. Process P_3 is executed before P_6, which gives the utility U_3 of 30, instead of 27.5 in the other case. Note that, after process P_1 is re-executed, we consider that no more faults can happen ($k = 1$). Thus, process P_6 does not experience any faults in the worst-case scenario depicted above the schedule for N_2. The overall utility produced in this case will be 92.5.

As can be seen, a variety of situations will occur during execution, which must be captured by the scheduler. We implement a quasi-static scheduling mechanism that generates a number of possible schedules off-line. The online scheduler will only perform the switching between alternative schedules, based on the pre-computed switching conditions. The alternative schedules will be selected depending on the occurrence of faults and on the actual execution time of processes. All schedules guarantee that hard deadlines are satisfied even in the worst case.

V. Scheduling

In this section we present our value-based quasi-static scheduling algorithm for fault-tolerant distributed systems.

A. Scheduling with Fault Tolerance

In this paper we will adapt the scheduling strategy which we have applied in [11] to generate fault-tolerant schedules for mixed soft and hard real-time monoprocessor systems. This strategy uses "recovery slack" in the schedule to accommodate the time needed for re-executions in case of faults. After each process P_i we assign a slack equal to $(t_i + \mu) \times f$, where f is the number of faults to tolerate, t_i is the worst-case execution time of the process and μ is the re-execution overhead. The slack is shared between processes in order to reduce the time allocated for recovering from faults. The advantage of this scheduling strategy is that it can quickly produce efficient fault-tolerant schedules that do not require much memory to support fault tolerance.

We extend the monoprocessor scheduling strategy [11] to capture communications on the bus. Moreover, in this work, we also improve the efficiency of our quasi-static strategy from [11], as discussed in Section V.C. We propose a greedy quasi-static scheduling heuristic that uses a fast and efficient simulator, which we have developed, inside the optimization loop.

In our scheduling approach for distributed hard real-time systems [9, 20] the sending and receiving times of messages on the bus are fixed and cannot be changed within one fault-tolerant schedule. Each message m_i, which is an output of process P_j is always scheduled at fixed time on the bus after the worst-case completion time of P_j. In this paper, we will refer to such a fault-tolerant multiprocessor schedule with recovery slacks and fixed communications as an f^N-schedule, where N is the number of computation nodes in the system.

B. Signalling and Schedules

Our primary objective is to maximize the total utility value of the application. However, pure f^N schedules with fixed sending times of messages can lower the total utility due to imposed communication restrictions. In this work, thus, we propose a *signalling* mechanism to overcome this restriction.

A *signalling message* is *broadcasted* by computation node N_j to all other nodes to inform if a certain condition has been satisfied on that node. In our case, this *condition* is the completion of process P_i on node N_j at such time that makes execution of another pre-calculated schedule f_n^N more beneficial than the presently running f_p^N schedule. The condition is *known* if it has been broadcasted to all computation nodes with the signalling message. Switching to the new schedule f_n^N is performed by all computation nodes in the system, after receiving the broadcasted signalling message with the corresponding "true" condition value. In the proposed signalling approach, we will send one signalling message for each process, *after the worst-case execution time for hard processes* or *after the expected execution time for soft processes* in the schedule. The number of signalling messages has been limited to one per process, taking into account problem complexity and in order to provide a simple yet efficient solution.

In Fig. 5, we consider application $\mathcal{A}_3$ composed of 5 processes, P_1 to P_5, where processes P_1 and P_5 are hard with deadlines of 170 and 400 ms, respectively, and processes P_2, P_3 and P_4 and all messages are soft. In Fig. 5a, a single f^2 schedule S_0 is generated, as depicted above the Gantt chart that outlines a possible execution scenario that would follow S_0. Message sending times on the bus are fixed and message m_2 is, thus, always sent at 150 ms (denoted as $m_2[150]$ in the schedule). m_2 has to wait for the worst-case completion time of P_2.[1] The order of processes is preserved on computation nodes. However, the start times of processes are varied, under the constraint that processes do not start earlier than their *earliest allowed start time* in the schedule. For example, process P_3 will wait until completion of process P_4 but will not start earlier than 160 ms, waiting for message m_2 (with the constraint denoted as $P_3(160)$ in the schedule). In schedule S_0 we also depict the number r of *allowed re-executions* for processes with the "+r" notation. For example, process P_2 is allowed to re-execute $r = 1$ time to tolerate

1. Note that if P_2 is re-executed, message m_2 will be, anyway, sent at 150 ms but will transport a "stale" value.

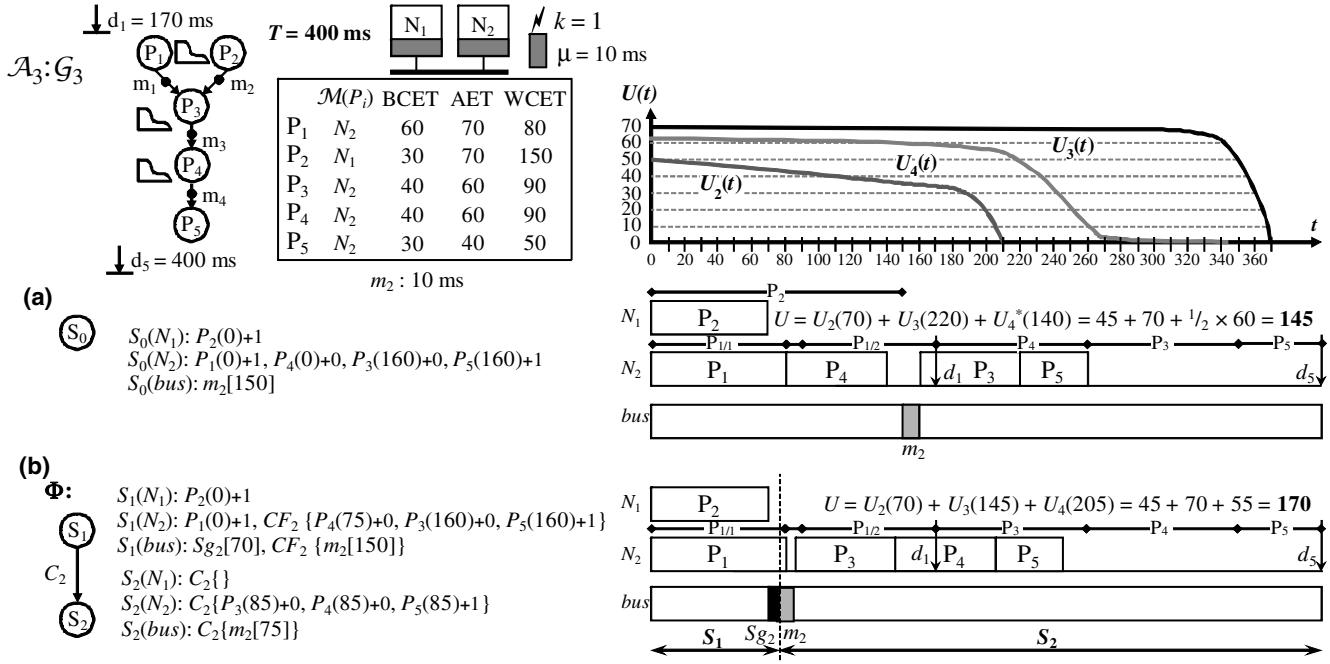


Figure 5. Signalling (1)

$k = 1$ fault, i.e., $P_2(0) + 1$, while processes P_3 and P_4 are not allowed to re-execute ($r = 0$).

In the execution scenario in Fig. 5a, which follows S_0, process P_2 completes at 70ms with utility of 45. Process P_4 is executed directly after process P_1 with resulting utility of $^1/_2 \times 60 = 30$. Process P_3 is waiting for message m_2 and completes with utility of 70. The overall utility is, thus, 145.

In the schedule shown in Fig. 5b, we employ the signalling mechanism. The signalling message Sg_2 is sent after the *expected execution time* of the *soft process* P_2, which can trigger the switch from the initial *root* schedule S_1 to the new schedule S_2. Schedule tree Φ, composed of schedules S_1 and S_2, is depicted above the Gantt chart in Fig. 5b. Switching between the schedules is performed upon known "true" condition C_2, which is broadcasted with the signalling message Sg_2. C_2 informs that P_2 has completed at such time that switching to S_2 has become profitable. In the opposite case, when switching to S_2 is not profitable, due to, for example, that P_2 has not completed at that time, Sg_2 transports a "false" condition CF_2. Thus, two execution scenarios are possible in the tree Φ: under "true" condition C_2 and under "false" condition CF_2. In schedule S_1, in the CF_2 case, processes P_3, P_4 and P_5 and message m_2 are grouped under this condition CF_2 as $CF_2\{P_4, P_3, P_5\}$ on node N_2 and as $CF_2\{m_2\}$ on the bus. They will be activated in the schedule only when "false" condition CF_2 is known. Respectively, in schedule S_2, onto which the scheduler switches in the case of "true" condition C_2, processes P_3, P_4 and P_5 and message m_2 are grouped under C_2 that will activate them.

With the signalling mechanism, overall utility of the execution of application $\mathcal{A}_3$ can be improved. In the Gantt chart in Fig. 5b, process P_2 completes at 70 ms with utility of 45. In this case switching to the schedule S_2 is profitable and, hence, signalling message Sg_2 with the "true" condition C_2 is broadcasted. The scheduler switches from S_1 to S_2 after arrival of Sg_2, as shown in the figure. According to the new order in

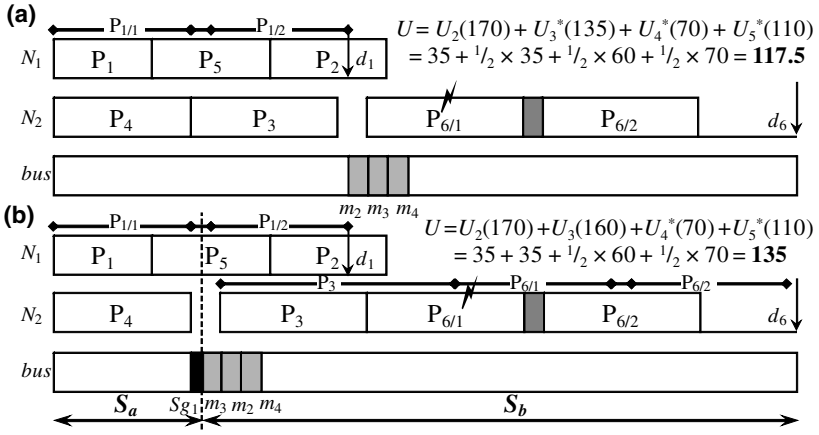


Figure 6. Signalling (2)

schedule S_2, process P_4 is executed after its predecessor P_3, with the utilities of 55 and 70, respectively. The resulting utility of this execution scenario, thus, will be 170, instead of only 145 in Fig. 5a.

In Fig. 6a we depict an execution scenario for application $\mathcal{A}_1$ from Fig. 4 on a single f^2 schedule. The schedule will produce an overall utility of 117.5 for this execution scenario. If, however, we send a signal Sg_1 after the worst-case execution time of the hard process P_1 and generate the corresponding schedule S_b, we can produce a much better overall utility of 135, by switching to the schedule S_b, as depicted in Fig. 6b.

C. Value-based Scheduling

We construct a tree of fault-tolerant schedules with the FTQuasiStatic heuristic outlined in Fig. 7. At first, we generate a *Root* f^N schedule with recovery slacks (line 1) using the GenerateFTSchedule heuristic, outlined in Fig. 8, which takes into account a service degradation rule $\mathcal{R}$. This schedule alone does not allow changing the order of processes and it fixes the sending times of messages on the bus. From *Root* we generate other schedules, to which the scheduler will switch to improve the overall utility, as discussed in Section IV. Signalling messages sent on the bus will trigger switches between these schedules, as discussed in Section V.B.

The quasi-static scheduling algorithm is iterating while improving the total utility value with new schedules (lines 3-20), at the same time, preserving deadlines of hard processes. At each iteration, we consider a *selected* schedule ϕ_{max}, which is initially the *Root* schedule (line 2) or has been obtained on the previous iteration (lines 14-18). We evaluate each process P_i in ϕ_{max} as completed with expected execution time, and generate a corresponding schedule ϕ_{new} (lines 8-13). After ϕ_{new} is generated, we determine when it is profitable to change to it from ϕ_{max}, and at what completion times of process P_i (line 10), such that deadlines of hard processes are satisfied, in the *interval partitioning step*. We heuristically compute *switching points* to ϕ_{new} by exploring possible completion times of process P_i in ϕ_{max} with a given interval ΔI. Then, we evaluate ϕ_{new} with its switching points by performing *simulation* of the tree with ϕ_{new} attached (line 11). Simulation is performed until simulation results converge with a given error probability $d\varepsilon$ and the total utility U_{new} is obtained, which also captures utility degradation due to "stale" values of soft processes in a variety of simulated execution scenarios. If ϕ_{new} improves the total utility, it is temporarily attached to the tree (added to set Φ' of temporarily selected schedules, line 12). During simulation, we run the actual online scheduler that executes a variety of execution scenarios. We are able to run 10.000s execution scenarios in a matter of seconds. All new schedules, which are generated, are evaluated and the schedule ϕ_{max}, which gives the best improvement in terms of total utility is *selected* (lines 14-18).[1] It is permanently added to the tree Φ and removed from the temporal set Φ' in the next iteration (line 7).

```
FTQuasiStatic(G, k, M, R): const Δl, dε
1  Root:= GenerateFTSchedule(∅, G, k, M, R)
2  Φ':= Root; Φ:= ∅; φmax:= Root
3  do
4    improvement:= false
5    if φmax ≠ ∅ then
6      improvement:= true
7      Φ:= Φ ∪ φmax; Remove(φmax, Φ')
8      for all Pi ∈ φmax do
9        φnew:= GenerateFTSchedule(Pi, φmax, G, k, M, R)
10       IntervalPartitioning(Pi, φmax, φnew, ΔI)
11       Unew:= Evaluate(Φ ∪ φnew, dε, R)
12       if Unew > Umax then Φ':= Φ' ∪ φnew
13     end for
14     Umax:= 0; φmax:= ∅
15     for all φj ∈ Φ' do
16       Uj:= Evaluate(Φ ∪ φj, dε, R)
17       if Umax < Uj then Umax:= Uj; φmax:= φj
18     end for
19   end if
20 while improvement
21 return Φ
end FTQuasiStatic
```

Figure 7. Quasi-Static Scheduling

The *Root* schedule and new schedules ϕ_{new} in the FTQuasiStatic algorithm (lines 1 and 9, Fig. 7) are generated with the GenerateFTSchedule heuristic in Fig. 8. This heuristic is based on the monoprocessor FTSS heuristic, which we have proposed in [11], with a number of differences such as handling communication over the bus with signalling messages and assigning *guards*, which is not straightforward in the context of static cyclic scheduling. Processes and messages in GenerateFTSchedule are scheduled under *known conditions* or guards [7]. The scheduler will switch from a parent schedule S_{parent} to the new schedule *FS* based on a "true" condition upon completion time of process P_i, transported by Sg_i, as discussed in Section V.B. *Current guard* is initially *true* in case of no parent schedule (and *FS* = *Root* schedule is generated, for example, S_1 in Fig. 5b). Otherwise, the initial value of *current guard* is that the given process P_i has completed at a certain time, which triggers "true" switching condition (line 1). Schedule *FS* begins after arrival of Sg_i with "true" condition (for example, S_2 begins after arrival of Sg_2 in Fig. 5b). In *FS*, the current guard will be updated after arrival of each signalling message Sg_j (for example, in S_1 in Fig. 5b the guard is updated to CF_2 after arrival of Sg_2 with "false" condition at 75 ms).

During construction of *FS*, we consider that P_i executes with its expected execution time for all execution scenarios in *FS*. Because of the reduced execution time of P_i, from the worst case to the expected case, more soft processes can be potentially scheduled in *FS* than in S_{parent}, which will give an increased overall utility to the schedule *FS* compared to S_{parent}.

The algorithm iterates while there exist processes in the ready list L_R, which are selected to be scheduled (lines 2-3). List L_R includes all unscheduled soft and "ready" hard processes. A "ready" hard process is an unscheduled hard process whose all hard predecessors have been scheduled. At each iteration, the heuristic determines which processes and messages, if chosen to be executed, lead to a schedulable solution. The "best"

1. Thus, we ensure *by construction* of the schedule tables that the hard deadlines are met, while we improve the utility of the tree by simulation.

processes on computation nodes and the "best" message on the bus are selected (line 4) that would potentially contribute to the greatest overall utility according to the MU priority function proposed in [4] and modified by us to capture possible process dropping [11]. During priority computation we consider the degraded utility $U^*(t)$, obtained with a service degradation rule $\mathcal{R}$, instead of the original $U(t)$. Out of the "best" processes and messages on different resources, we select the one which can be scheduled the earliest (line 5).

We schedule the selected process or message *Bst* under the current set of known conditions *K* (lines 6-7). For each process we add recovery slack, as discussed in Section V.A (AddSlack, line 8). Recovery slacks of hard processes will accommodate all *k* faults. Recovery slacks of soft processes, however, must not reduce the utility value of the no fault scenarios and will, thus, accommodate as much faults as possible but, in general, not all *k* faults. For each process P_j we also add its signalling message Sg_j to the ready list L_R (AddSgMsg, line 8). When scheduling a signalling message (line 9), we change current guards of computation nodes at arrival time of the message.

Finally, the list L_R of ready nodes is updated with the ready successors of the scheduled process or message (line 10). Those soft processes whose executions will not lead to any utility increase or would exceed application period *T*, are removed from the ready list L_R, i.e., dropped (line 11).

The GenerateFTSchedule heuristic will return either *FS* = *Root* or a fault-tolerant subschedule *FS* that will be integrated into the quasi-static tree Φ by the FTQuasiStatic heuristic.

D. Switching between Schedules

Switching between schedules in the quasi-static tree in run-time is performed very fast. At each possible switching point, e.g. after completion of a process P_i, the scheduler can have at most two possible alternatives, i.e., to signal or not to signal the switching to the "new" schedule. We store a pointer to this "new" schedule in its parent schedule, when attaching the "new" schedule to the quasi-static tree (line 7, Fig. 7). The pointer is associated to a signalling message of the process P_i, whose finishing time triggers the potential switching to the "new" schedule. We also store the pre-computed switching intervals as attached to this process. Thus, the online scheduler will only check if the completion time of the process P_i matches the switching interval (i.e., an "if" statement has to be executed in the scheduler code) and, if so, will encapsulate the corresponding switching condition into the signalling message, which is broadcasted through the bus.

```
GenerateFTSchedule(P_i, S_parent, G, k, M, R)
1  SetCurrentGuard(P_i, FS)
2  L_R:= GetReadyNodes(G)
3  while L_R ≠ ∅ do
4    for each resource r_j ∈ {N, B} do
                Bst_j:= MMUSelect(r_j, L_R, R)
5    Bst:= SelectBestCRT(all Bst_j)
6    K:= ObtainGuards(Bst, FS)
7    Schedule(Bst, K, FS)
8    if Bst is a process then
              AddSlack(Bst, FS); AddSgMsg(Bst, L_R)
9    if Bst is a signal then
              UpdateCurrentGuards(Bst, FS)
10   UpdateReadyNodes(Bst, L_R)
11   Dropping(r_j, L_R)
12 end while
13 return FS
end GenerateFTSchedule
```

Figure 8. Schedule Generation

Upon arrival of the signalling message, the online scheduler on each computation node will de-reference the pointer, associated to this message, and switch to the "new" schedule, which has been pre-generated and optimized exactly for this situation by the algorithms in Figs. 7-8. Thus, no searching of the right schedules is performed online and the online time overhead is practically negligible. We have implemented and evaluated our schedule switching mechanism as a part of our simulator, which has demonstrated its efficiency.

VI. Experimental Results

For our experiments we have generated 1008 applications of 20, 30, 40, 50, 60, 70 and 80 processes (24 applications for each dimension) for 2, 3, 4, 5, 6 and 7 computation nodes. Execution times of processes in each application have been varied from 10 to 100 ms, and message transmission times between 1 and 4 ms. Deadlines and utility functions have been assigned individually for each process in the application. For the main set of experiments we have considered that 50% of all processes in each application are soft and the other 50% are hard. We have set the maximum number *k* of transient faults to 3 and recovery overhead to 10% of process execution time. As worst-case execution time of a process can be much longer than its expected execution time, we have assigned a tail factor T_f = WCET / (AET × 2) to each process. Experiments have been run on a 2.83 GHz Intel Pentium Core2 Quad processor with 8 Gb memory.

At first, we were interested to evaluate our heuristics with the increased number of computation nodes and the number of processes. For this set of experiments we have set the tail factor to 5 for all soft processes in the applications. The table in Fig. 9a shows an improvement of our quasi-static scheduling on top of a single f^N schedule in terms of total utility, considering an f^N schedule in the case of no faults as a 100% baseline. The average improvement is ranging from 3% to 22% in case of 20 processes on 6 computation nodes and 40 processes on 2 computation nodes, respectively. Note that in this table we depict the normalized utility values for quasi-static scheduling that have been obtained in the case of *no faults*. During our experiments, we have observed that the utility values for faulty cases closely follow the no fault scenarios with about 1% decrease (on average) in the utility value for each fault, as illustrated, for example, in Fig. 9c.

As the number of computation nodes increases, in general, our quasi-static scheduling improves less on top of a single f^N schedule, which could seem counterintuitive. However, this is because, with an increased number of computation nodes, less soft processes are allocated per node on average and the number

of possible value-based intra-processor scheduling decisions by the quasi-static scheduling is reduced. At the same time, the number of inter-processor scheduling decisions is supposed to increase. However, these decisions are less crucial from the point of view of obtaining utility due to the greater availability of computational resources, and a single f^N schedule with a fixed order of processes is sufficient to utilize them. Moreover, a limited number of signalling messages, e.g. one per process, restricts the number of possible inter-processor decisions by the quasi-static scheduling. Hence, the total number of exploitable scheduling alternatives is reduced with more computation nodes. In case of 20 processes and 2 nodes, the average number of soft processes is 5 per node (considering 50% soft processes in the application, $20 \times 0.5 / 2 = 5$) and the utility improvement is 17%. In the case of 7 nodes, only an average of 1.4 soft processes are allocated per node and, hence, the utility improvement is only 3%. However, as the number of processes in the application is growing, the trend is softened. For 40 processes and 2 nodes the improvement is 22% with the average of 10 soft processes per node. For 7 nodes the improvement is 9% with 2.9 soft processes per node on average. For 80 processes, the improvement in case of 2 and 7 nodes is already the same, 12% for 20 and 5.7 soft processes per node, respectively.

The average size of the quasi-static tree is between 2.2 schedules for 20 processes on 6 nodes and 8.0 schedules for 60 processes on 5 nodes that correspond to 1.3Kb and 9.4Kb of memory, respectively. As the size of the application grows, the amount of memory per schedule will increase. Thus, a tree of 7 schedules for an 80-process application on 5 nodes would require 11.4Kb of memory. During our experiments we could also observe that for many applications already 2-3 schedules give at least half of the utility improvement. For example, for 40 processes and 2 computation nodes, 3 schedules will give 21% improvement. To compare, in our previous approach [11] for applications with 30 processes in the monoprocessor context, we needed at least 8 schedules in the tree to achieve the same level of improvement.

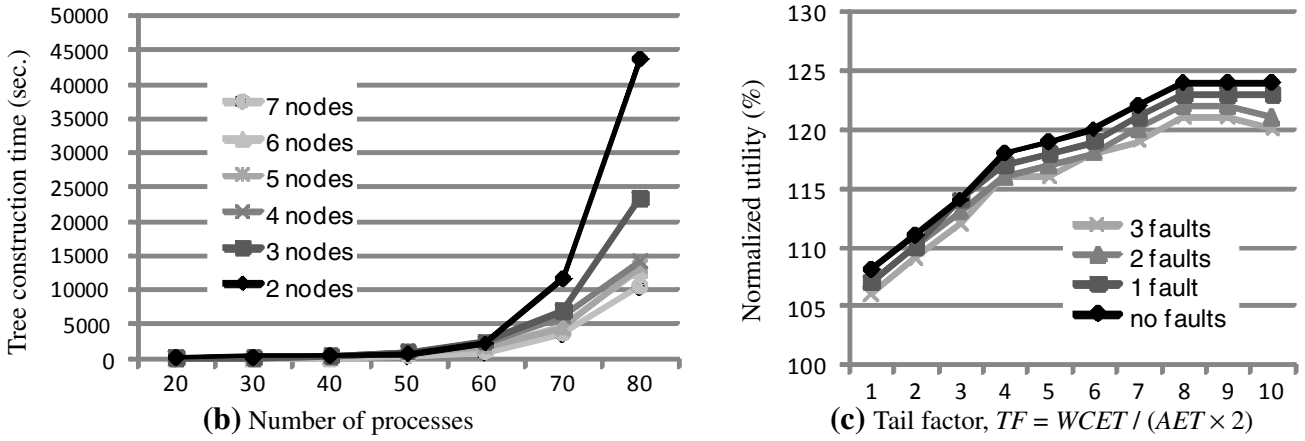
Off-line tree construction times with FTQuasiStatic are depicted in Fig. 9b. Our heuristic produces a tree of schedules in a matter of minutes for 20, 30 and 40 processes, and below one hour for 50 and 60 processes. Tree construction time is around 4 hours for 5 nodes and 80 processes. Although the off-line tree construction time is high for large applications, the online overhead is still very small and is constant for all applications, in spite of the application size, as discussed in Section V.D.

For a given total number of processes, the tree construction time of the quasi-static algorithm is reduced with the number of computation nodes. This can be explained taking into account that the average number of soft processes per computation node is reduced, which leads to a smaller amount of scheduling alternatives to be explored by the quasi-static scheduling. In case of 2 nodes, a greater number of soft processes for each computation node is allocated and, as the result, more valuable scheduling alternatives in total have to be considered than in the case of, for example, 7 nodes. Our scheduling algorithm, thus, has to spend much more time evaluating scheduling alternatives for less nodes than in the case of more nodes, as confirmed by the experimental results given in Fig. 9b.

In our next set of experiments, we have varied the percentage of soft and hard processes in the applications. We have ad-

(a) Normalized utility ($U_n = U_{FTQ.S.}/U_{f^N} \times 100\%$) and the number of schedules (n)

N	20 proc.		30 proc.		40 proc.		50 proc.		60 proc.		70 proc.		80 proc.	
	U_n	n	U_n	n	U_n	n	U_n	n	U_n	n	U_n	n	U_n	n
2	117	4.3	119	5.3	122	5.6	117	5.8	116	5.4	114	6.5	112	4.9
3	111	3.8	113	4.6	119	6.9	119	6.0	118	5.8	115	7.4	114	7.5
4	109	2.7	112	4.5	113	5.4	118	7.0	115	7.0	115	7.3	113	6.8
5	106	2.5	112	2.6	113	5.6	112	5.8	116	8.0	113	5.8	115	7.0
6	103	2.2	109	4.2	110	4.5	112	6.1	115	7.3	113	6.7	113	6.3
7	103	1.8	106	3.0	109	4.8	112	5.4	110	6.0	110	5.8	112	7.0



(b) Number of processes

(c) Tail factor, $TF = WCET / (AET \times 2)$

Figure 9. Experimental Results

ditionally generated 1080 applications of 20, 30, 40, 50 and 60 processes for 2, 3, 4, 5 and 6 computation nodes, respectively. The percentage of soft processes has been initially set to 10% and hard processes to 90%. We have gradually increased the percentage of soft processes up to 90% of soft processes and 10% of hard processes, with a step of 10% (and have generated 120 applications of different dimensions for each setup). The improvement produced by FTQuasiStatic over all dimensions is between 14 and 16%. The average number of schedules has increased from 2 for 10% of soft processes to 6 for 90% of soft processes, with the increased execution time of the heuristic from 10 to 30 minutes.

We were also interested to evaluate our heuristic with different tail factors T_f, which we have varied from 1 to 10 for applications with 40 processes on 2 nodes. As our results in Fig. 9c show, the improvement produced by FTQuasiStatic is larger in the case of a greater tail factor. If for the tail factor 1, the improvement is 7.5%, it is around 20% for the tail factors of 5-6 and 24% for the tail factors above 7. Note that the total utilities of fault scenarios, as depicted in Fig. 9c, closely follow the no fault scenarios with only about 1% utility value decrease for each fault. In our previous approach [11] in the monoprocessor context, the impact of faults was more significant, between 3 to 16% utility decrease for each fault.

We have also run our experiments on a real-life example, a vehicle cruise controller (CC) [20], composed of 32 processes mapped on 3 computation units: Electronic Throttle Module (ETM), Anti-lock Breaking System (ABS) and Transmission Control Module (TCM). We have considered that 16 processes implement critical functions and are hard. The rest of processes have been assigned with utility functions. The tail factor is 5 with $k = 2$ transient faults, recovery overhead of 10% of process execution time and signalling message transmission time of 2 ms. In terms of total utility FTQuasiStatic could improve on top of a single f^N schedule with 22%, with 4 schedules that would need 4.8Kb of memory.

VII. Conclusions

We have presented an approach to scheduling of mixed soft and hard real-time distributed systems with fault-tolerance. A quasi-static scheduling heuristic is used to generate a tree of fault-tolerant schedules off-line that maximizes the total utility value of the application and satisfies deadlines. Depending on the current execution situation and fault occurrences, an online scheduler, with low online overhead, chooses which schedule to execute, based on the pre-computed switching conditions. The obtained tree of schedules can deliver an increased level of quality-of-service and guarantee timing constraints of embedded applications under limited amount of resources.

References

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

[2] V. B. Balakirsky and A. J. H. Vinck, "Coding Schemes for Data Transmission over Bus Systems", *In Proc. IEEE Intl. Symp. on Information Theory*, 1778-1782, 2006.

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

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

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

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

[7] P. Eles, A. Doboli, P. Pop, and Z. Peng, "Scheduling with Bus Access Optimization for Distributed Embedded Systems", *IEEE Trans. on VLSI*, 8(5), 472-491, 2000.

[8] 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.

[9] 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.

[10] V. Izosimov, "Scheduling and Optimization of Fault-Tolerant Distributed Embedded Systems", *Ph.D. Thesis No. 1290, Dept. of Computer and Information Science, Linköping University*, 2009.

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

[12] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Synthesis of Flexible Fault-Tolerant Schedules with Preemption for Mixed Soft and Hard Real-Time Systems", *11th Euromicro Conf. on Digital System Design*, 71-80, 2008.

[13] M. Jonsson and K. Kunert, "Reliable Hard Real-Time Communication in Industrial and Embedded Systems", *Proc. 3rd IEEE Intl. Symp. on Industrial Embedded Systems*, 184-191, 2008.

[14] 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.

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

[16] 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.

[17] 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.

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

[19] E. Piriou, C. Jego, P. Adde, R. Le Bidan, and M. Jezequel, "Efficient Architecture for Reed Solomon Block Turbo Code", *In Proc. IEEE Intl. Symp. on Circuits and Systems (ISCAS)*, 4 pp., 2006.

[20] P. Pop, V. Izosimov, P. Eles, and Z. Peng, "Design Optimization of Time- and Cost-Constrained Fault-Tolerant Embedded Systems With Checkpointing and Replication", *IEEE Trans. on VLSI*, 17(3), 389-402, 2009.

[21] 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.

[22] 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.

[23] 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 CAD*, 25(1), 111–125, 2006.