HAL
open science

Design Optimization of Time-and Cost-Constrained Fault-Tolerant Distributed Embedded Systems

Viacheslav Izosimov, Paul Pop, Petru Eles, Zebo Peng

To cite this version:

Viacheslav Izosimov, Paul Pop, Petru Eles, Zebo Peng. Design Optimization of Time-and Cost-Constrained Fault-Tolerant Distributed Embedded Systems. DATE'05, Mar 2005, Munich, Germany. pp.864-869. hal-00181227

HAL Id: hal-00181227

https://hal.science/hal-00181227v1

Submitted on 23 Oct 2007

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Design Optimization of Time- and Cost-Constrained Fault-Tolerant Distributed Embedded Systems

Viacheslav Izosimov, Paul Pop, Petru Eles, Zebo Peng
Computer and Information Science Dept., Linköping University, Sweden
{paupo|petel|zebpe|viaiz}@ida.liu.se

Abstract

In this paper we present an approach to the design optimization of fault-tolerant embedded systems for safety-critical applications. Processes are statically scheduled and communications are performed using the time-triggered protocol. We use process re-execution and replication for tolerating transient faults. Our design optimization approach decides the mapping of processes to processors and the assignment of fault-tolerant policies to processes such that transient faults are tolerated and the timing constraints of the application are satisfied. We present several heuristics which are able to find fault-tolerant implementations given a limited amount of resources. The developed algorithms are evaluated using extensive experiments, including a real-life example.

1. Introduction

Safety-critical applications have to function correctly and meet their timing constraints even in the presence of faults. Such faults can be permanent (i.e., damaged microcontrollers or communication links), transient (e.g., caused by electromagnetic interference), or intermittent (appear and disappear repeatedly). The transient faults are the most common, and their number is continuously increasing due to the continuously raising level of integration in semiconductors.

Researchers have proposed several hardware architecture solutions, such as MARS [14], TTA [15] and XBW [4], that rely on hardware replication to tolerate a single permanent fault in any of the components of a fault-tolerant unit. Such approaches, can be used for tolerating transient faults as well, but they incur very large hardware cost. An alternative to such purely hardware-based solutions are approaches such as re-execution, replication, checkpointing.

Pre-emptive on-line scheduling environments are flexible enough to handle such fault-tolerance policies. Several researchers have shown how the schedulability of an application can be guaranteed at the same time with appropriate levels of fault-tolerance [1, 2, 9, 21]. 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 [13].

The disadvantage of static scheduling approaches, however, is their lack of flexibility, which makes it difficult to integrate tolerance towards unpredictable fault occurrences. Thus, researchers have proposed approaches for integrating fault-tolerance into the framework of static scheduling. A simple heuristic for combining together several static schedules in order to mask fault-patterns through replication is proposed in [5], without considering the timing constraints of the application. This approach is used as the basis for cost and fault-tolerance trade-offs within the Metropolis environment [16]. Graph transformations are used in [3] in order to introduce replication mechanisms into an application. Such a graph transformation approach, however, does not work for re-execution, which has to be considered during the construction of the static schedules.

Fohler [7] proposes a method for joint handling of aperiodic and periodic processes by inserting slack for aperiodic processes in the static schedule, such that the timing constraints of the periodic processes are guaranteed. In [8] he equates the aperiodic processes with fault-tolerance techniques that have to be invoked on-line in the schedule table slack to handle faults. Overheads due to several fault-tolerance techniques, including replication, re-execution and recovery blocks, are evaluated.

When re-execution is used in a distributed system, Kandasamy [11] proposes a list-scheduling technique for building a static schedule that can mask the occurrence of faults, thus making the re-execution transparent. Slacks are inserted into the schedule in order to allow the re-execution of processes in case of faults. The faulty process is re-executed, and the processor switches to a contingency schedule that delays the processes on the corresponding processor, making use of the slack introduced. The authors propose an algorithm for reducing the necessary slack for re-execution. This algorithm has later been applied to the fault-tolerant transmission of messages on a time-division multiple-access bus (TDMA) [12].

Applying such fault-tolerance techniques introduces overheads in the schedule and thus can lead to unschedulable systems. Very few researchers [11, 16] consider the optimization of implementations to reduce the overheads due to fault-tolerance and, even if optimization is considered, it is very limited and does not include the concurrent usage of several fault-tolerance techniques. Moreover, the application of fault-tolerance techniques is considered in isolation, and thus is not reflected at all levels of the design process, including mapping, scheduling and bus access optimization. In addition, the communication aspects are not considered or very much simplified.

In this paper, we consider hard real-time safety-critical applications mapped on distributed embedded systems. Both the processes and the messages are scheduled using static cyclic scheduling. The communication is performed using a communication environment based on the time-triggered protocol [14]. We consider two distinct fault-tolerance techniques: re-execution of processes, which provides time-redundancy, and active replication, which provides space-redundancy. We show 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. We propose several optimization algorithms for the mapping of processes to processors and the assignment of fault-tolerance techniques to processes such that the application is schedulable and no additional hardware resources are necessary.

The next two sections present the system architecture and the application model, respectively. Section 4 introduces the design optimization problems tackled, and Section 5 proposes a tabu-search based algorithm for solving these problems. The evaluation of the proposed approaches, including a real-life example consisting of a cruise controller are presented in Section 6. The last section presents our conclusions.

2. System Architecture

2.1 Hardware Architecture and Fault Model

We consider architectures composed of a set $\mathcal{N}$ nodes which share a broadcast communication channel. Every *node* $N_i \in \mathcal{N}$ consists, among others, of a communication controller and a CPU. Figure 1a depicts an architecture consisting of four nodes.

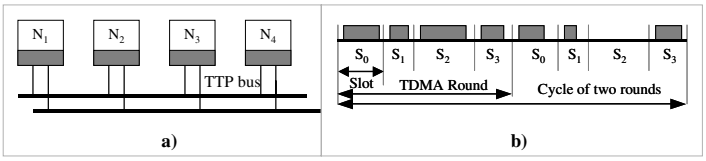

Figure 1. System Architecture Example

Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE'05)
1530-1591/05 $ 20.00 IEEE

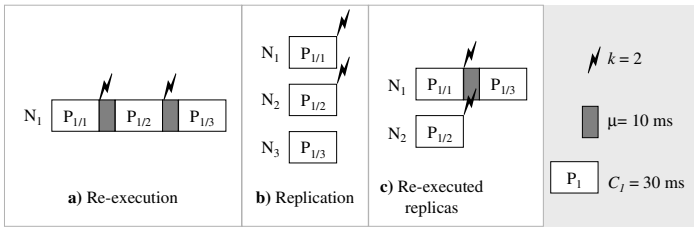

Figure 2. Worst-Case Fault Scenario and Fault-Tolerant Techniques

The communication controllers implement the protocol services and run independently of the node's CPU. We consider the time-triggered protocol (TTP) [14] as the communication infrastructure for a distributed real-time system. However, the research presented is also valid for any other TDMA-based bus protocol that schedules the messages statically based on a schedule table like, for example, the SAFEbus [10] protocol used in the avionics industry.

The TTP has a replicated bus that integrates all the services necessary for fault-tolerant real-time systems. According to the TTP, each node N_i can transmit only during a predetermined time interval, the so called TDMA slot S_i, see Figure 1b. In such a slot, a node can send several messages packed in a frame. A sequence of slots corresponding to all the nodes in the TTC is called a TDMA round. A node can have only one slot in a TDMA round. Several TDMA rounds can be combined together in a cycle that is repeated periodically. The TDMA access scheme is imposed by a message descriptor list (MEDL) that is located in every TTP controller. The MEDL serves as a schedule table for the TTP controller which has to know when to send/receive a frame to/from the communication channel.

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. We have generalized the fault-model from [11] that assumes that one single transient fault may occur on any of the nodes in the system during the application execution. In our model, we consider that at most k transient faults[1] may occur anywhere in the system during one operation cycle of the application. Thus, not only several transient faults may occur simultaneously on several processors, but also several faults may occur on the same processor. We consider that the transient faults can have a worst-case duration of μ, from the moment the fault is detected until the system is back to its normal operation, and that a fault is confined to a single process and does not affect other processes.

2.2 Software Architecture and Fault-Tolerance Techniques

We have designed a software architecture which runs on the CPU in each node, and which has a real-time kernel as its main component. The processes are activated based on the local schedule tables, and messages are transmitted according to the MEDL. For more details about the software architecture and the message passing mechanism the reader is referred to [6].

The error detection and fault-tolerance mechanisms are part of the software architecture. We assume a combination of hardware-based (e.g., watchdogs, signature checking) and software-based error detection methods, systematically applicable without any knowledge of the application (i.e., no reasonableness and range checks) [4]. We also assume that all faults can be found using such detection methods, i.e., no byzantine faults which need voting on the output of replicas for detection. The software architecture, including the real-time kernel, error detection and fault-tolerance mechanisms are themselves fault-tolerant.

We use two mechanisms for tolerating faults: re-execution and active replication. Let us consider the example in Figure 2, where we have process P_1 and a fault-scenario consisting of $k = 2$ faults with a duration μ = 10 ms that can happen during one cycle of operation. In Figure 2a we have the worst-case fault scenario for re-execution, when the first fault happens at the end of the process P_1's execution. The fault is detected and, after an interval μ, P_1 can be re-executed. Its second execution is labeled with $P_{1/2}$, which, in the worst-case could also experience a fault at the end. Finally, the third re-execution of P_1, namely $P_{1/3}$, will execute without error. In the case of active replication, depicted in Figure 2b, each replica is executed on a different processor. Three replicas are needed to tolerate the two possible faults and, in the worst-case scenario depicted in Figure 2b, only the execution of $P_{1/3}$ is successful. In addition, we consider a third case, presented in Figure 2c, which combines re-execution and replication for tolerating faults in a process. In this case, for tolerating the two faults we use two replicas and one re-execution: the process $P_{1/1}$, which has $P_{1/2}$ as a replica, is re-executed.

With active replication, the input has to be distributed to all the replicas. Since we do not consider the type of faults that need replica agreement, our execution model assumes that the descendants of replicas can start as soon as they have received the first valid message from a replica. Replica determinism is achieved as a by-product of the underlying TTP architecture [17].

3. Application Model

We model an application $\mathcal{A}$ as a set of directed, acyclic, polar graphs $\mathcal{G}(\mathcal{V}, \mathcal{E}) \in \mathcal{A}$. 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 can be activated after all its inputs have arrived and it issues its outputs when it terminates. The communication time between processes mapped on the same processor is considered to be part of the process worst-case execution time and is not modeled explicitly. Communication between processes mapped to different processors is performed by message passing over the bus. Such message passing is modeled as a communication process inserted on the arc connecting the sender and the receiver process.

The combination of fault-tolerance policies to be applied to each process is given by two functions. $\mathcal{F}_R$: $\mathcal{V} \rightarrow \mathcal{V}_R$ determines which processes are replicated. When active replication is used for a process P_i, we introduce several replicas into the process graph $\mathcal{G}$, and connect them to the predecessors and successors of P_i.. The second function $\mathcal{F}_X$: $\mathcal{V} \cup \mathcal{V}_R \rightarrow \mathcal{V}_X$ applies re-execution to the processes in the application, including to the replicas in $\mathcal{V}_R$, if necessary, see Figure 2c. Let us denote the tuple $<\mathcal{F}_R, \mathcal{F}_X>$ with $\mathcal{F}$.

The mapping of a process graph $\mathcal{G}$ is given by a function $\mathcal{M}$: $\mathcal{V} \cup \mathcal{V}_R \rightarrow \mathcal{N}$, where $\mathcal{N}$ is the set of nodes in the architecture. For a process $P_i \in \mathcal{V} \cup \mathcal{V}_R$, $\mathcal{M}(P_i)$ is the node to which P_i is assigned for execution. Each process P_i can potentially be mapped on several nodes. Let $\mathcal{N}_{P_i} \subseteq \mathcal{N}$ be the set of nodes to which P_i can potentially be mapped. We consider that for each $N_k \in \mathcal{N}_{P_i}$, we know the worst-case execution time $C_{P_i}^{N_k}$ of process P_i, when executed on N_k. We also consider that the size of the messages is given.

All processes and messages belonging to a process graph $\mathcal{G}_i$ have the same period $T_i = T_{\mathcal{G}_i}$ which is the period of the process graph. A deadline $D_{\mathcal{G}_i} \leq T_{\mathcal{G}_i}$ is imposed on each process graph $\mathcal{G}_i$. In addition, processes can have associated individual release times and deadlines. If communicating processes are of different periods, they are combined into a hyper-graph capturing all process activations for the hyper-period (LCM of all periods).

4. Design Optimization Problems

In this paper, by policy assignment we denote the decision whether a certain process should be re-executed or replicated. Mapping a process means placing it on a particular node in the architecture.

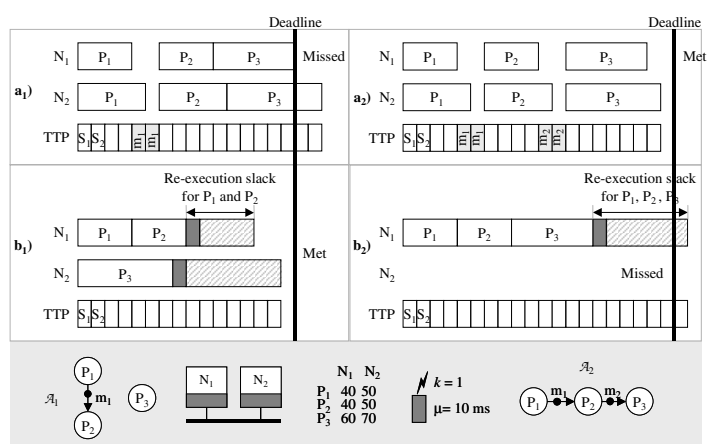

Figure 3. Comparison of Replication and Re-Execution

1. The number of faults k can be larger than the number of processors in the system.

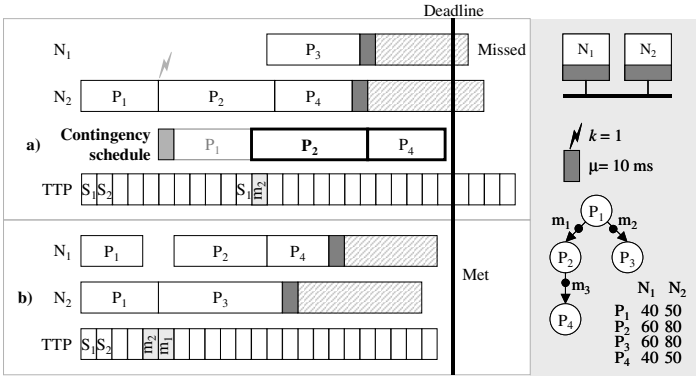

Figure 4. Combining Re-execution and Replication

There could be cases where the policy assignment decision is taken based on the experience and preferences of the designer, considering aspects like the functionality implemented by the process, the required level of reliability, hardness of the constraints, legacy constraints, etc. We denote with $\mathcal{P}_R$ the subset of processes which the designer has assigned replication, while $\mathcal{P}_X$ contains processes which are to be re-executed.

Most processes, however, do not exhibit certain particular features or requirements which obviously lead to re-execution or replication. Let $\mathcal{P}$ be the set of processes in the application $\mathcal{A}$. The subset $\mathcal{P}^+ = \mathcal{P} \setminus (\mathcal{P}_X \cup \mathcal{P}_R)$ of processes could use any of the two techniques for tolerating faults. Decisions concerning the policy assignment to this set of processes can lead to various trade-offs concerning, for example, the schedulability properties of the system, the amount of communication exchanged, the size of the schedule tables, etc.

For part of the processes in the application, the designer might have already decided their mapping. For example, certain processes, due to constraints like having to be close to sensors/actuators, have to be physically located in a particular hardware unit. They represent the set $\mathcal{P}_M$ of already mapped processes. Consequently, we denote with $\mathcal{P}^* = \mathcal{P} \setminus \mathcal{P}_M$ the processes for which the mapping has not yet been decided.

Our problem formulation is as follows:

- As an input we have an application $\mathcal{A}$ given as a set of process graphs (Section 3) and a system with a set of nodes $\mathcal{N}$ connected using the TTP.
- The fault model is given by the parameters k and μ, which denote the total number of transient faults that can appear in the system during one cycle of execution and their duration, respectively.
- As introduced previously, $\mathcal{P}_X$ and $\mathcal{P}_R$ are the sets of processes for which the fault-tolerance policy has already been decided. Also, $\mathcal{P}_M$ denotes the set of already mapped processes.

We are interested to find a system configuration ψ such that the k transient faults are tolerated and the imposed deadlines are guaranteed to be satisfied, within the constraints of the given architecture $\mathcal{N}$.

Determining a system configuration $\psi = <\mathcal{F}, \mathcal{M}, \mathcal{S}>$ means:

1. finding a combination of fault-tolerance policies $\mathcal{F}$ for each processes in $\mathcal{P}^+ = \mathcal{P} \setminus (\mathcal{P}_R \cup \mathcal{P}_X)$;
2. deciding on a mapping $\mathcal{M}$ for each processes in $\mathcal{P}^* = \mathcal{P} \setminus \mathcal{P}_M$;
3. deriving the set $\mathcal{S}$ of tables for each processor and the MEDL for TTP.

4.1 Fault-Tolerance Policy Assignment

Let us illustrate some of the issues related to policy assignment. In the example presented in Figure 3 we have the application $\mathcal{A}_1$ with three processes, P_1 to P_3, and an architecture with two nodes, N_1 and N_2. The worst-case execution times on each node are given in a table to the right of the architecture. Note that N_1 is faster than N_2. The fault model assumes a single fault, thus $k = 1$, with a duration $\mu = 10$ ms. The application $\mathcal{A}_1$ has a deadline of 160 ms depicted with a thick vertical line. We have to decide which fault-tolerance technique to use. In Figure 3 we depict the schedules[1] for each node, and for the TTP bus. Node N_1 is allowed to transmit in slot S_1, while node N_2 can use slot S_2. A TDMA round is formed of slot S_1 followed by slot S_2, each of 10 ms length. Comparing the schedules in Figure $3a_1$ and $3b_1$, we can observe that using (a_1) active replication the deadline is missed. In order to guarantee that time constraints are satisfied in the presence of faults, re-execution slacks have to finish before the deadline. However, using (b_1) re-execution we are able to meet the deadline. However, if we consider application $\mathcal{A}_2$ with process P_3 data dependent on P_2, the deadline is missed in Figure $3b_2$ if re-execution is used, and it is met when replication is used as in Figure $3a_2$.

Note that in Figure $3b_1$ processes P_1 and P_2 can use the same slack for re-execution. Similarly, in Figure $3b_2$, one single slack of size $C_3 + \mu$ is enough to tolerate one fault in any of the processes. In general, re-execution slacks can be shared as long as they allow a re-execution of processes to tolerate faults.

This example shows that the particular technique to use, has to be carefully adapted to the characteristics of the application. Moreover, the best result is most likely to be obtained when both techniques are used together, some processes being re-executed, while others replicated. Let us consider the example in Figure 4, where we have an application with four processes mapped on an architecture of two nodes. In Figure 4a all processes are re-executed, and the depicted schedule is optimal for re-execution.

We use a particular type of re-execution, called transparent re-execution [11], that hides fault occurrences on a processor from other processors. On a processor N_i where a fault occurs, the scheduler has to switch to a contingency schedule that delays descendants of the faulty process. However, a fault happening on another processor, is not visible on N_i, even if the descendants of the faulty process are mapped on N_i. For example, in order to isolate node N_1 from the occurrence of a fault in P_1 on node N_2, message m_2 from P_1 to P_3 cannot be transmitted at the end of P_1's execution. Message m_2 has to arrive at the destination even in the case of a fault occurring in P_1, so that P_3 can be activated on node N_2 at a fixed start time, regardless of what happens on node N_1, i.e., transparently. Consequently, m_2 can only be transmitted after a time $C_1 + \mu$ passes, at the end of the potential re-execution of P_1, depicted in grey. Message m_2 is delivered in the slot S_2 of the TDMA round corresponding to node N_2. With this setting, using re-execution will miss the deadline. Once a fault happens, the scheduler in N_2 will have to switch to a contingency schedule, depicted with thick-border rectangles.

However, combining re-execution with replication, as in Figure 4b where process P_1 is replicated, will meet the deadline. In this case, message m_2 does not have to be delayed to mask the failure of process P_1. Instead, P_2 and P_3 will have to receive m_1 and m_2, respectively, from both replicas of P_1, which will introduce a delay due to the inter-processor communication on the bus.

4.2 Mapping and Bus Access Optimization

For a distributed system, the communication infrastructure has an important impact on the mapping decisions [20]. Not only is the mapping influenced by the protocol setup, but the fault-tolerance policy assignment cannot be done separately from the mapping design task. Consider the example in Figure 5. Let us suppose that we have applied a mapping algorithm without considering the fault-tolerance aspects, and we have

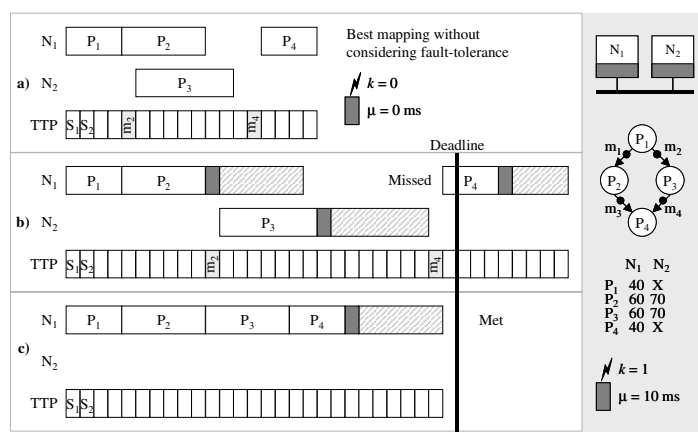

Figure 5. Mapping and Fault-Tolerance

1. The schedules depicted are optimal.

IEEE COMPUTER SOCIETY

obtained the best possible mapping, depicted in Figure 5a. If we apply on top of this mapping a fault-tolerance technique, for example, re-execution as in Figure 5b, we miss the deadline. The re-execution has to be considered during the mapping process, and then the best mapping will be the one in Figure 5c which clusters all processes on the same processor in order to reduce the re-execution slack and the delays due to the masking of faults.

In this paper, we will consider the assignment of fault-tolerance policies at the same time with the mapping of processes to processors. However, to simplify the presentation we will not discuss the optimization of the communication channel. Such an optimization can be performed with the techniques we have proposed in [19] for non fault-tolerant systems.

5. Design Optimization Strategy

The design problem formulated in the previous section is NP complete. Our strategy is outlined in Figure 6 and has three steps:

1. In the first step (lines 1–3) we decide very quickly on an initial bus access configuration $\mathcal{B}^0$, and an initial fault-tolerance policy assignment $\mathcal{F}^0$ and mapping $\mathcal{M}^0$. The initial bus access configuration (line 1) is determined by assigning nodes to the slots ($S_i = N_i$) and fixing the slot length to the minimal allowed value, which is equal to the length of the largest message in the application. The initial mapping and fault-tolerance policy assignment algorithm (InitialMPA line 2 in Figure 6) assigns a re-execution policy to each process in $\mathcal{P}^+$ and produces a mapping for the processes in $\mathcal{P}^*$ that tries to balance the utilization among nodes and buses. The application is then scheduled using the ListScheduling algorithm outlined in Section 5.1. If the application is schedulable the optimization strategy stops.
2. The second step consists of a greedy heuristic GreedyMPA (line 4), discussed in Section 5.2, that aims to improve the fault-tolerance policy assignment and mapping obtained in the first step.
3. If the application is still not schedulable, we use, in the third step, a tabu search-based algorithm TabuSearchMPA presented in Section 5.2. Finally, the bus access optimization is performed.

If after these three steps the application is unschedulable, we conclude that no satisfactory implementation could be found with the available amount of resources.

5.1 List Scheduling

Once a fault-tolerance policy and a mapping are decided, as well as a communication configuration is fixed, the processes and messages have to be scheduled. We use a list scheduling algorithm for building the schedule tables for the processes and deriving the MEDL for messages.

Before applying list scheduling, we merge the application graphs into one single merged graph Γ, as detailed in [18], with a period equal to the LCM of all constituent graphs. List scheduling heuristics are based on priority lists from which processes are extracted in order to be scheduled at certain moments. A process P_i is placed in the ready list $\mathcal{L}$ if all its predecessors have been already scheduled. All ready processes from the list $\mathcal{L}$ are investigated, and that process P_i is selected for placement in the schedule which has the highest priority. We use the modified partial critical path priority function presented in [6]. At the same time with placing processes in the schedule, the messages are also scheduled using the ScheduleMessage function from [6]. The ListScheduling loops until the ready list $\mathcal{L}$ is empty.

During scheduling, re-execution slack is introduced in the schedule for the re-executed processes. The introduction of re-execution slack is discussed in [11] where the total amount of slack is reduced through slack-sharing, as depicted in Figure 3b$_2$, where processes P_1 to P_3 can share the

OptimizationStrategy($\mathcal{A}$, $\mathcal{N}$)
1 **Step 1:** $\mathcal{B}^0$ = InitialBusAccess($\mathcal{A}$, $\mathcal{N}$)
2 ψ^0 = InitialMPA($\mathcal{A}$, $\mathcal{N}$, $\mathcal{B}^0$)
3 **if** $\mathcal{S}^0$ is schedulable **then** stop **end if**
4 **Step 2:** ψ= GreedyMPA($\mathcal{A}$, $\mathcal{N}$, ψ^0)
5 **if** $\mathcal{S}$ is schedulable **then** stop **end if**
6 **Step 3:** ψ = TabuSearchMPA($\mathcal{A}$, $\mathcal{N}$, ψ)
7 **return** ψ
end OptimizationStrategy

Figure 6. The General Strategy

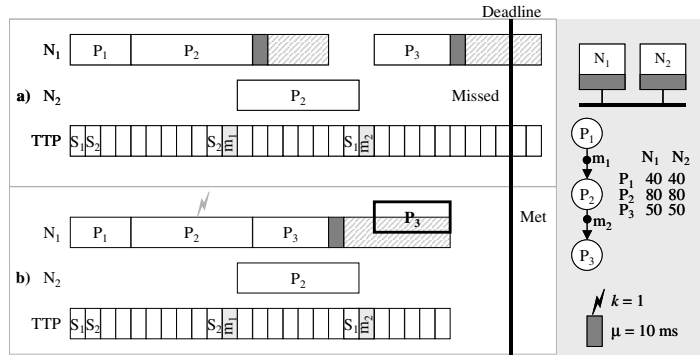

Figure 7. Scheduling Replica Descendants

same slack for re-execution in the case of a fault.

However, the notion of "ready process" in [11] is different for us in the case of processes waiting inputs from replicas. In that case, a process can be placed in the schedule as soon as we are certain that at least one valid message has arrived from a replica. Let us consider the example in Figure 7, where P_2 is replicated. In the worst-case fault-scenario, P_2 on processor N_1 can fail, and thus P_3 has to receive message m_2 from the P_2 replica on processor N_2. Thus, P_3 has to be placed in the schedule as in Figure 7a. However, our scheduling algorithm will place P_3 as in Figure 7b instead, immediately following P_2 on N_1. In addition, it will create a contingency schedule for P_3 on processor N_1, as depicted in Figure 7b using a rectangle with a thicker margin. The scheduler on N_1 will switch to this schedule only in the case of an error occurring in P_2 on processor N_1. This contingency schedule has two properties. First, P_3 starts such that the arrival of m_2 from the P_2's replica on N_2 is guaranteed. Up to this point, it looks similar to the case in Figure 7a, where P_3 has been started at this time from the beginning. However, the contingency schedule has another important property: although P_3's failure is handled through re-execution, the contingency schedule will not contain any re-execution slack for P_3. That is because, according to the fault model, no more errors can happen. Thus, the deadline is met, even if any of the processes will experience a fault.

5.2 Mapping and Fault-Policy Assignment

For deciding the mapping and fault-policy assignment we use two steps, see Figure 6. One is based on a greedy heuristic, GreedyMPA. If this step fails, we use in the next step a tabu search approach, TabuSearchMPA.

Both approaches investigate in each iteration all the processes on the critical path of the merged application graph Γ, and use design transformations (moves) to change a design such that the critical path is reduced. Let us consider the example in Figure 8, where we have an application of four processes that has to tolerate one fault, mapped on an architecture of two nodes. Let us assume that the current solution is the one depicted in Figure 8a. In order to generate neighboring solutions, we perform design transformations that change the mapping of a process, and/or its fault-tolerance policy. Thus, the neighbor solutions generated starting from Figure 8a, are the solutions presented in Figure 8b–8e. Out of these, the solution is Figure 8c is the best in terms of schedule length.

The greedy approach selects in each iteration the best move found and applies it to modify the design. The disadvantage of the greedy approach is that it can "get stuck" into a local optima. To avoid this, we have implemented a tabu search algorithm, presented in Figure 9.

The tabu search takes as an input the merged application graph Γ, the architecture $\mathcal{N}$ and the current implementation ψ, and produces a schedulable and fault-tolerant implementation x^{best}. The tabu search is based on a neighborhood search technique, and thus in each iteration it generates the set of moves N^{now} that can be reached from the current solution x^{now} (line 7 in Figure 9). In our implementation, we only consider changing the mapping or fault-tolerance policy of the processes on the critical path, denoted with *CP* in Figure 9. We define the critical path as the path through the merged graph Γ which corresponds to the longest delay in the schedule table. For example, in Figure 8a, the critical path is formed from P_1, m_2 and P_3.

IEEE COMPUTER SOCIETY

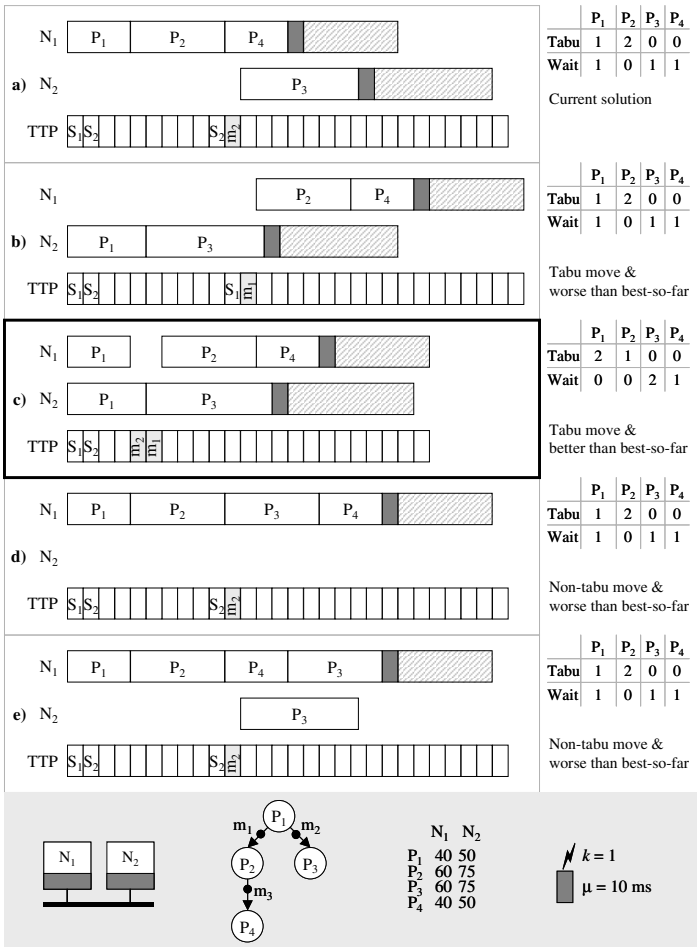

Figure 8. Moves and Tabu History

The key feature of a tabu search is that the neighborhood solutions are modified based on a selective history of the states encountered during the search. The selective history is implemented in our case through the use of two tables, *Tabu* and *Wait*. Each process has an entry in this tables. If $Tabu(P_i)$ is non-zero, it means that the process is "tabu", i.e., should not be selected for generating moves, while if $Wait(P_i)$ is greater than the number of processes in the graph, $|\Gamma|$, the process has waited a long time and should be selected for diversification. Thus, lines 9 and 10 of the algorithm, a move will be removed from the neighborhood solutions if it is tabu. However, tabu moves are also accepted if they are better than the best-so-far solution (line 10). In line 12 the search is diversified with moves which have waited a long time without being selected.

In lines 14–20 we select the best one out of these solutions. We prefer a solution that is better than the best-so-far x^{best} (line 17). If such a solution does not exist, then we choose to diversify. If there are no diversification moves, we simply choose the best solution found in this iteration, even if it is not better than x^{best}. Finally, the algorithm updates the best-so-far solution, and the selective history tables *Tabu* and *Wait*. The algorithm ends when a schedulable solutions has been found, or an imposed time-limit has been reached.

Figure 8 illustrates how the algorithm works. Let us consider that the current solution x^{now} is the one presented in Figure 8a, with the corresponding selective history presented to its right, and the best-so-far solution x^{best} is the one in Figure 4a. The generated solutions are presented in Figure 8b–8e. The solution (b) is removed from the set of considered solutions because it is tabu, and it is not better than x^{best}. Thus, solutions (c)–(e) are evaluated in the current iteration. Out of these, the solution in Figure 8c is selected, because although it is tabu, it is better than x^{best}. The table is updated as depicted to the right of Figure 8c in bold, and the iterations continue with solution (c) as the current solution.

6. Experimental Results

For the evaluation of our algorithms we used applications of 20, 40, 60, 80, and 100 processes (all unmapped and with no fault-tolerance policy assigned) implemented on architectures consisting of 2, 3, 4, 5, and 6 nodes, respectively. We have varied the number of faults depending on the architecture size, considering 3, 4, 5, 6, and 7 faults for each architecture dimension, respectively. The duration μ of a fault has been set to 5 ms. Fifteen examples were randomly generated for each application dimension, thus a total of 75 applications were used for experimental evaluation. We generated both graphs with random structure and graphs based on more regular structures like trees and groups of chains. Execution times and message lengths were assigned randomly using both uniform and exponential distribution within the 10 to 100 ms, and 1 to 4 bytes ranges, respectively. The experiments were done on Sun Fire V250 computers.

We were first interested to evaluate the proposed optimization strategy in terms of overheads introduced due to fault-tolerance. Hence, we have implemented each application, on its corresponding architecture, using the OptimizationStrategy (MXR) strategy from Figure 6. In order to evaluate MXR, we have derived a reference non-fault tolerant implementation, NFT. The NFT approach is an optimized implementation similar to MXR, but we have removed the moves that decide the fault-tolerance policy assignment. To the NFT implementation thus obtained, we would like to add fault-tolerance with as little as possible overhead, and without adding any extra hardware resources. For these experiments, we have derived the shortest schedule within an imposed time limit: 10 minutes for 20 processes, 20 for 40, 1 hour for 60, 2 hours and 20 min. for 80 and 5 hours and 30 min. for 100 processes.

The first results are presented in Table 1a, where we have four columns: the third column presents the average overheads introduced by MXR compared to NFT, while the second and fourth columns present the maximum and minimum overhead, respectively. Let δ_{MXR} and δ_{NFT} be the schedule lengths obtained using MXR and NFT, respectively. The overhead is defined as $100 \times (\delta_{MXR} - \delta_{NFT}) / \delta_{NFT}$. We can see that the overheads due to fault-tolerance grow with the application size. MXR approach can offer fault-tolerance within the constraints of the architecture at an average overhead of approximately 100%. However, even for applications of 60 processes, there are cases where the overhead is as low as 52%.

We were also interested to evaluate our MXR approach in the case the number of faults and their length varies. We have considered applications with 60 processes mapped on four processors, and we have varied the number k of faults from 2, 4, 6, 8, to 10, using a constant μ = 5 ms. Table 1b shows that the overheads increase substantially as the number of faults that have to be tolerated increase. This is to be expected, since we need

```
TabuSearchMPA(Γ, 𝒩, ψ)
1   -- given a merged application graph Γ and an architecture 𝒩 produces a policy
2   -- assignment ℱ and a mapping ℳ such that Γ is fault-tolerant & schedulable
3   x^best = x^now = ψ; BestCost = ListScheduling(Γ, 𝒩, x^best) -- Initialization
4   Tabu = ∅; Wait = ∅ -- The selective history is initially empty
5   while x^best not schedulable ∧ TimeLimit not reached do
6      -- Determine the neighboring solutions considering the selective history
7      CP = CriticalPath(Γ); N^now = GenerateMoves(CP)
8      -- eliminate tabu moves if they are not better than the best-so-far
9      N^tabu = {move(P_i) | ∀ P_i ∈ CP ∧ Tabu(P_i)=0 ∧ Cost(move(P_i)) < BestCost}
10     N^non-tabu = N \ N^tabu
11     -- add diversification moves
12     N^waiting = {move(P_i) | ∀ P_i ∈ CP ∧ Wait(P_i) > |Γ|}
13     N^now = N^non-tabu ∪ N^waiting
14     -- Select a solution based on aspiration criteria
15     x^now = SelectBest(N^now);
16     x^waiting = SelectBest(N^waiting); x^non-tabu = SelectBest(N^non-tabu)
17     if Cost(x^now) < BestCost then x = x^now -- select x^now if better than best-so-far
18        else if ∃ x^waiting then x = x^waiting -- otherwise diversify
19        else x = x^non-tabu -- if no better and no diversification, select best non-tabu
20     end if
21     -- Perform selected move
22     PerformMove(x); Cost = ListScheduling(Γ, 𝒩, x)
23     -- Update the best-so-far solution and the selective history tables
24     If Cost < BestCost then x^best = x; BestCost = Cost end if
25     Update(Tabu); Update(Wait)
26  end while
27  return x^best
end TabuSearchMPA
```

Figure 9. The Tabu Search Algorithm

IEEE COMPUTER SOCIETY

(a) Application size					(b) Number of faults 60 procs., μ=5				(c) μ 20 procs., k=3			
procs.	*k*	%max	%avg.	%min	*k*	%max	%avg.	%min	μ	%max	%avg.	%min
20	3	98.36	70.67	48.87	2	52.44	32.72	19.52	1	78.69	57.26	34.29
40	4	116.77	84.78	47.30	4	110.22	76.81	46.67	5	95.90	70.67	48.87
60	5	142.63	99.59	51.90	6	162.09	118.58	81.69	10	122.95	89.24	67.58
80	6	177.95	120.55	90.70	8	250.55	174.07	117.84	15	132.79	107.26	75.82
100	7	215.83	149.47	100.37	10	292.11	219.79	154.93	20	149.01	125.18	95.60

Table 1. Overheads of MXR compared to NFT

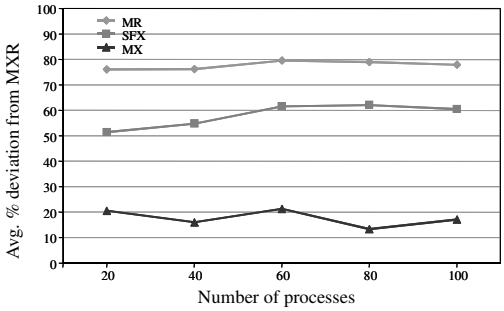

Figure 10. Comparing MXR with MX, MR and SFX

more replicas and/or re-executions if there are more faults. Similarly, we have kept the number of faults constant to 3, and varied μ: 1, 5, 10 15 and 20 ms, for 20 processes and two processors. We can observe in Table 1c that the overhead also increases with the increase in fault duration. However, the increase due to the fault duration is significantly lower compared to the increase due to the number of faults.

As a second set of experiments, we were interested to evaluate the quality of our MXR optimization approach. Thus, together with the MXR approach we have also evaluated two extreme approaches: MX that considers only re-execution, and MR which relies only on replication for tolerating faults. MX and MR use the same optimization approach as MRX, but besides the mapping moves, they consider assigning only re-execution or replication, respectively. In Figure 10 we present the average percentage deviations of the MX and MR from MXR in terms of overhead. We can see that by optimizing the combination of re-execution and replication, MXR performs much better compared to MX and MR. On average, MXR is 77% and 17.6% better than MR and MX, respectively. There are also situations, for graphs with 60 processes, for example, where MXR is able to reduce the overhead with up to 40% compared to MX and 90% compared to MR. This shows that considering re-execution at the same time with replication can lead to significant improvements.

In Figure 10 we have also presented a straightforward strategy SFX, which first derives a mapping without fault-tolerance considerations (using MXR without fault-tolerance moves) and then applies re-execution. This is a solution that can be obtained by a designer without the help with our fault-tolerance optimization tools. We can see that the overheads thus obtained are very large compared to MXR, up to 77% more on average. This shows that the optimization of the fault-tolerance policy assignment has to be addressed at the same time with the mapping of functionality. In Figure 10 we also see that replication (MR) is worst than even the straightforward re-execution (SFX). However, by carefully optimizing the usage of replication alongside re-execution (MXR), we are able to obtain results that are significantly better than re-execution only (MX).

Finally, we considered a real-life example implementing a vehicle cruise controller (CC). The process graph that models the CC has 32 processes, and is described in [18]. The CC was mapped on an architecture consisting of three nodes: Electronic Throttle Module (ETM), Anti-lock Breaking System (ABS) and Transmission Control Module (TCM). We have considered a deadline of 250 ms, $k = 2$ and $\mu = 2$ ms.

In this setting, the MRR produced a schedulable fault-tolerant implementation with a worst-case system delay of 229 ms, and with an overhead compared to NFT of 65%. If only one policy is used for fault-tolerance, as in the case of MX and MR, with 253 and 301 ms, respectively, the deadline is missed.

7. Conclusions

In this paper we have addressed the optimization of distributed embedded systems for fault-tolerance hard real-time applications. The processes are scheduled with static cyclic scheduling, while for the message transmission we use the TTP. We have employed two fault-tolerance techniques for tolerating faults: re-execution, which provides time-redundancy, and active replication, which provides space-redundancy.

We have implemented a tabu search-based optimization approach that decides the mapping of processes to the architecture and the assignment of fault-tolerance policies to processes. Our main contribution is that we have considered the interplay of fault-tolerance techniques for reducing the overhead due to fault-tolerance. As our experiments have shown, by carefully optimizing the system implementation we are able to provide fault-tolerance under limited resources.

References

[1] A. Bertossi, L. Mancini, "Scheduling Algorithms for Fault-Tolerance in Hard-Real Time Systems", *Real Time Systems*, 7(3), 229–256, 1994.

[2] A. Burns et al., "Feasibility Analysis for Fault-Tolerant Real-Time Task Sets", *Euromicro Workshop on Real-Time Systems*, 29–33, 1996.

[3] P. Chevochot, I. Puaut, "Scheduling Fault-Tolerant Distributed Hard-Real Time Tasks Independently of the Replication Strategies", *Real-Time Computing Systems and Applications Conf.*, 356–363, 1999.

[4] V. Claeson, S. Poldena, J. Söderberg, "The XBW Model for Dependable Real-Time Systems", *Parallel and Distributed Systems Conf.*, 1998.

[5] C. Dima et al, "Off-line Real-Time Fault-Tolerant Scheduling", *Euromicro Parallel and Distributed Processing Workshop* , 410–417, 2001.

[6] P. Eles et al., "Scheduling with Bus Access Optimization for Distributed Embedded Systems", *IEEE Transactions on VLSI Systems,* 8(5), 472-491, 2000.

[7] G. Fohler, "Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems", *IEEE Real-Time Systems Symposium*, 152–161, 1995.

[8] G. Fohler, "Adaptive Fault-Tolerance with Statically Scheduled Real-Time Systems", *Euromicro Real-Time Systems Workshop,* 161–167, 1997.

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

[10] K. Hoyme, K. Driscoll, "SAFEbus", *IEEE Aerospace and Electronic Systems Magazine*, 8(3), 34–39, 1992.

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

[12] N. Kandasamy, J. P. Hayes B.T. Murray "Dependable Communication Synthesis for Distributed Embedded Systems," *Computer Safety, Reliability and Security Conf.*, 275–288, 2003.

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

[14] H. Kopetz et al., "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach", *IEEE Micro*, 9(1), 25–40, 1989.

[15] H. Kopetz, Günter Bauer, "The Time-Triggered Architecture", *Proceedings of the IEEE*, 91(1), 112–126, 2003.

[16] 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.

[17] S. Poldena, *Fault Tolerant Systems–The Problem of Replica Determinism*, Kluwer Academic Publishers, 1996.

[18] P. Pop, "Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems", *Ph. D. Thesis No. 833, Dept. of Computer and Information Science, Linköping University,* 2003.

[19] P. Pop, P. Eles, Z. Peng, "Schedulability Analysis and Optimization for the Synthesis of Multi-Cluster Distributed Embedded Systems", *Design, Automation and Test in Europe Conference and Exhibition*, pp. 184-189, 2003.

[20] P. Pop et al., "Design Optimization of Multi-Cluster Embedded Systems for Real-Time Applications", *Design, Automation and Test in Europe Conference and Exhibition*, pp. 1028-1033, 2004.

[21] Y. Zhang, K. Chakrabarty, "Energy-Aware Adaptive Checkpointing in Embedded Real-Time Systems", *DATE Conf.*, 918–923, 2003.