Synthesis of Fault-Tolerant Schedules with Transparency/Performance Trade-offs for Distributed Embedded Systems

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

Abstract

In this paper we present an approach to the scheduling of fault-tolerant embedded systems for safety-critical applications. Processes and messages are statically scheduled, and we use process re-execution for recovering from multiple transient faults. If process recovery is performed such that the operation of other processes is not affected, we call it transparent recovery. Although transparent recovery has the advantages of fault containment, improved debugability and less memory needed to store the fault-tolerant schedules, it will introduce delays that can violate the timing constraints of the application. We propose a novel algorithm for the synthesis of fault-tolerant schedules that can handle the transparency/performance trade-offs imposed by the designer, and makes use of the fault-occurrence information to reduce the overhead due to fault tolerance. We model the application as a conditional process graph, where the fault occurrence information is represented as conditional edges and the transparent recovery is captured using synchronization nodes.

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 increasing due to the raising level of integration in semiconductors.

Researchers have proposed several hardware architecture solutions, such as MARS [13], TTA [14] and XBW [3], 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 if the number of transient faults is larger than one. An alternative to such purely hardware-based solutions are approaches such as re-execution, replication, checkpointing.

Several researchers have shown how the schedulability of an application can be guaranteed at the same time with appropriate levels of fault-tolerance using pre-emptive online scheduling [1, 2, 8, 18]. Considering their high degree of predictability, researchers have proposed approaches for integrating fault-tolerance into the framework of static scheduling [12]. A simple heuristic for combining several static schedules in order to mask fault-patterns through replication is proposed in [4], without, however, considering any timing constraints. This approach is used as the basis for cost and fault-tolerance trade-offs within the Metropolis environment [15].

Fohler [5] 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 [6] 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 [10] proposes a list-scheduling technique for building a static schedule that can mask the occurrence of faults, 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 an alternative 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 [11].

In [9] we have shown how re-execution and active replication can be combined in an optimized implementation that leads to a schedulable fault-tolerant application without increasing the amount of employed resources. There, we have extended the scheduling algorithm in [10] to produce the fault-tolerant schedules for a given combination of fault-tolerant policies. We have considered *transparent re-execution*, where a fault occurring on one processor is masked to the other processors in the system, i.e., the recovery in case of a fault is *transparent*. Such fault masking has several advantages: provides fault-containment, improves debugability, and reduces the memory required for storing the fault-tolerant schedules. However, its main disadvantage is that it introduces delays into the schedule, needed to mask fault occurrences, which can lead to timing constraints violations.

In this paper, we consider a very different trade-off, namely, transparency versus performance and memory. We propose a fine-grained approach to transparency, by handling fault-containment at the application-level instead of resource-level, thus offering the designer the possibility to *trade-off transparency for performance*.

We propose a novel algorithm for the synthesis of fault tolerant schedules that can handle the transparency/performance trade-offs imposed by the designer. Our approach makes use of the fault-occurrence information to reduce the overhead due to fault tolerance in order to fulfill the timing constraints. We use a *fault-tolerant conditional process graph* (FT-CPG) to model the application: *conditional edges* are used for modelling fault occurrences, while *synchronization nodes* capture the fine-grained transparency requirements. The synthesis problem is formulated as a FT-CPG scheduling problem. The proposed algorithm not only handles fine-grained transparency, but, as the experimental results will show, also significantly outperforms the existing approach [9] in terms of the quality of the produced schedules.

The next two sections present the system architecture and the application model, respectively. Section 4 highlights the importance of supporting transparency/performance trade-offs. Section 5 introduces the FT-CPG model, and Section 6 presents the proposed FT-CPG scheduling algorithm. The evaluation of the proposed approaches, including a real-life example is presented in Section 7.

2. System Model

We consider architectures composed of a set $\mathcal{N}$ of nodes which share a broadcast communication channel. The communication channel is statically scheduled such that one node at a time has access to the bus, according to the schedule determined off-line.

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.

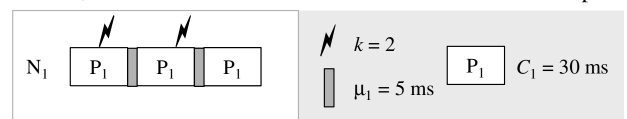


Figure 1. Re-execution

The processes activation and message transmission is done based on the local schedule tables.

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 [10] 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 may occur anywhere in the system during one operation cycle of the application. The number of faults can be larger than the number of processors in the system. Several transient faults may occur simultaneously on several processors as well as several faults may occur on the same processor.

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) [3]. The *error detection overhead* is considered as part of the process worst-case execution time. We 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. In addition, we assume that message fault-tolerance is achieved at the communication level, for example through hardware replication of the bus.

We use re-execution for tolerating faults. Let us consider the example in Fig. 1, where we have process P_1 and a fault-scenario consisting of $k = 2$ transient faults that can happen during one cycle of operation. In the worst-case fault scenario depicted in Fig. 1, the first fault happens during the process P_1's first execution, and is detected by the error detection mechanism. After a worst-case *recovery overhead* of $\mu_1 = 5$ ms, depicted with a light gray rectangle, P_1 will be executed again. Its second execution in the worst-case could also experience a fault. Finally, the third execution of P_1 will take place without fault.

3. Application Model

We model an application $\mathcal{A}(\mathcal{V}, \mathcal{E})$ as a set of directed, acyclic, polar graphs $\mathcal{G}_i(\mathcal{V}_i, \mathcal{E}_i) \in \mathcal{A}$. 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. Processes are non-preemptable and thus cannot be interrupted during its execution. Fig. 2 depicts an application $\mathcal{A}$ consisting of a graph $\mathcal{G}_1$ with four processes, P_1 to P_4.

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, and is depicted with black dots in the graph in Fig. 2.

The mapping of a process in the application is determined by a function $\mathcal{M}$: $\mathcal{V} \rightarrow \mathcal{N}$, where $\mathcal{N}$ is the set of nodes in the architecture. For a process $P_i \in \mathcal{V}$, $\mathcal{M}(P_i)$ is the node to which P_i is assigned for execution. We consider that the mapping is given, and we know the worst-case execution time C_{P_i} of process P_i, when executed on $\mathcal{M}(P_i)$. We also consider that the size of the messages is given. In Fig. 2, the mapping is given in the table besides the application graph.

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. Transparency/Performance Trade-offs

Although transparent recovery has the advantages of fault containment, improved debugability and less memory needed to store the fault-tolerant schedules, it will introduce delays that can violate the timing constraints of the application. These delays can be reduced by trading-off transparency for performance.

Let us consider the example in Fig. 2, where we have an application consisting of four processes, P_1 to P_4 and three messages, m_1 to m_3, mapped on an architecture with two processors, N_1 and N_2. Messages m_1 and m_2 are sent from P_1 to processes P_4 and P_3, respectively. Message m_3 is sent from P_2 to P_3. The worst-case execution times of each process are depicted in the table, and the deadline of the application is 210 ms. We consider a fault scenario where two transient faults ($k = 2$) can occur.

Whenever a fault occurs, the faulty process has to be re-executed. Thus, the scheduler in a processor that experiences a fault has to switch to another schedule containing a different start time for that process. For example, according to the schedule in Fig. 2a$_1$, processes are scheduled at times indicated by the white rectangles in the Gantt chart. Once a fault occurs in P_3, the scheduler on node N_1 will have to switch to another schedule, where P_3 is delayed with $C_3 + \mu$ to account for the fault. If, during the execution of P_3, a second fault occurs, the scheduler has to switch to another schedule illustrated in Fig. 2a$_2$.

All the alternative schedules needed to run the application in case of faults are produced off-line by the scheduling algorithm. The end-to-end worst-case delay of an application is given by the maximum finishing time of any schedule, since this is a situation that can happen in the worst-case scenario. For the application in Fig. 2a$_1$, the largest delay is produced by the schedule depicted in Fig. 2a$_2$, which has to be activated when two faults happen in P_3.

In Fig. 2 we illustrate four alternative scheduling strategies, representing different transparency/performance trade-offs. For each alternative, on the left side (a$_1$–d$_1$) we show the schedule when no faults occur, while the right side (a$_2$–d$_2$) depicts the corresponding worst-case scenario, resulting in the longest schedule. Thus, we would like to have in a$_2$–d$_2$ schedules that meet the deadline of 210 ms depicted with a thick vertical line.

Depending on how the schedule table is constructed, the re-execution of a process has a certain impact on the execution of other processes. In Fig. 2a$_1$, we have constructed the schedule such that each execution of a process P_i is followed by a *recovery slack*, which is idle time on the processor, needed to recover (re-execute) the failed process. For example, for P_3 on node N_2, we introduce a recovery slack of $k \times (C_3 + \mu) =$ 50 ms to make sure that we can recover P_3 even in the case it experiences the maximum number faults (Fig. 2a$_2$). Thus, a fault occurrence that leads to the re-execution of any process P_i will impact only P_i. We call such an approach *fully transparent* because fault occurrence in a process is transparent to *all* other processes on the same or other processors.

However, this approach has the drawback of introducing unnecessarily large delays into the schedule table. The end-to-end delay in this case is 265 ms, corresponding to the schedule in Fig. 2a$_2$, which will miss the deadline. The straightforward way to reduce the end-to-end delay is to share the re-execution slacks among several processes [9]. For example, in Fig. 2b$_1$, processes P_1 and P_2 share the same re-execution slack on processor N_1. This shared slack has to be large enough to accommodate the recovery of the largest process (in our case P_1) in the case of k faults. This slack can then handle k faults also in P_2, which takes less to execute than P_1.

In Fig. 2b we consider *transparent recovery*, where the fault occurring on one processor is masked to the other processors in the system but can impact processes on the same processor. This is the approach that we have used for scheduling in [9] where we focused only on the optimization of fault-tolerance policy assignment. On a processor N_i where a fault occurs, the scheduler has to switch to an alternative schedule that delays descendants of the faulty process running on the same processor N_i. However, a fault happening on another processor is not visible on N_i,

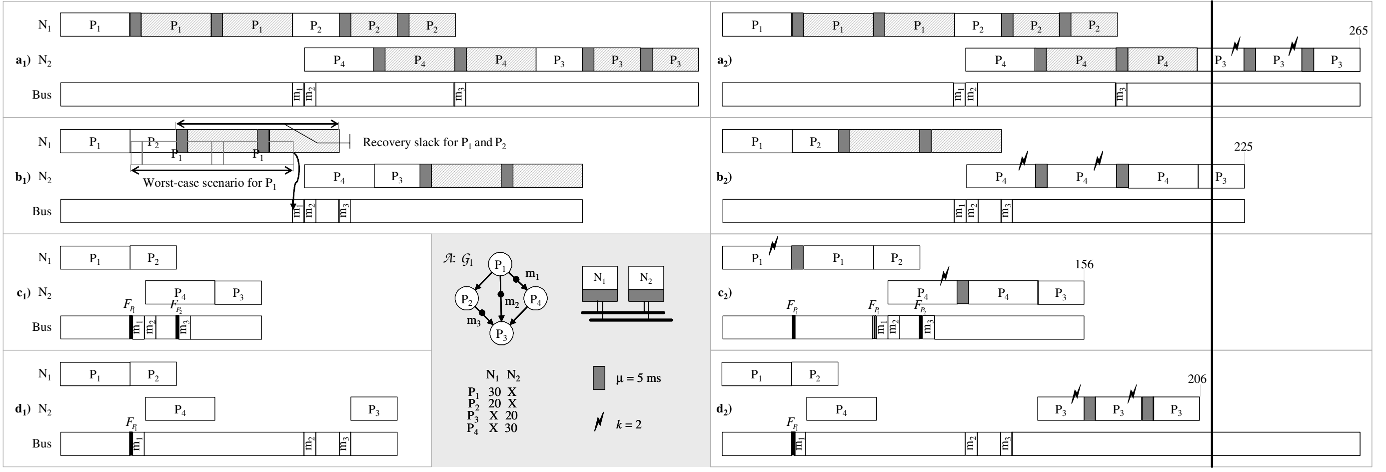


Figure 2. Transparency/Performance Trade-offs

even if the descendants of the faulty process are mapped on N_i. For example, in Fig. 2b$_1$, where we assume that no faults occur, in order to isolate node N_2 from the occurrence of a fault on node N_1, messages m_1 and m_2 from P_1 to P_4 and P_3, respectively, cannot be transmitted at the end of P_1's execution. Messages m_1 and m_2 have to arrive at the destination at a fixed time, regardless of what happens on node N_1, i.e., transparently. Consequently, the messages can only be transmitted after a time $k \times (C_1 + \mu)$, at the end of the recovery of P_1 in the worst-case scenario. However, a fault in P_1 will delay process P_2 which is on the same processor. This approach will lead to a reduced delay of 225 ms (still greater than the deadline) for the worst-case scenario, which corresponds to two faults happening in P_4, as depicted in Fig. 2b$_2$.

To meet the deadline, another approach, depicted in Fig. 2c, is not to mask fault occurrences at all. In this case, even the processes on the other processors will be affected by the fault occurrence. For example, the information about a fault occcurence in P_1 on N_1 will have to be sent to processor N_2 in order to switch to an alternative schedule that delays the scheduling of P_4, which receives message m_1 from P_1. This is done via the error message F_{P_1}, depicted as a black rectangle on the bus, which broadcasts the error occurrence on P_1 to other processors (see Section 6 for details). This would lead to a worst-case scenario of only 156 ms, depicted in Fig. 2c$_2$, that meets the deadline.

However, transparency (masking fault occurrences) is highly desirable for the reasons outlined in the introduction of this section, and a designer would like to introduce as much transparency as possible without violating the timing constraints. Thus, a more fine-grained approach to transparency is required. Such an approach is depicted in Fig. 2d, where faults are transparent to process P_3 and its input messages m_2 and m_3, but not to P_1, P_2, P_4 and m_1. In this case, P_3, m_2 and m_3 are said to be *frozen*, i.e., they have the same start time in all the schedules. The debugability is improved because it is easier to observe the behavior of P_3 in the alternative schedules. Its start time does not change due to the handling of faults. Moreover, the memory needed to store the alternative schedules is also improved with transparency, since there are less start times to store. In this case, the end-to-end delay of the application is 206, as depicted in Fig. 2d$_2$, and the deadline is met.

In this paper, we propose a fine-grained approach to transparency offering the designer the possibility to trade-off transparency for performance. Given an application $\mathcal{A}(\mathcal{V}, \mathcal{E})$ we will capture the transparency using the function $\mathcal{T}: \mathcal{V} \rightarrow$ *frozen*, where $v_i \in \mathcal{V}$ is a node in the application graph, which can be either a process or a communication message. In a fully transparent system, all messages and processes are frozen. A system with transparent recovery has all the inter-processor messages frozen. Our approach allows the designer to specify the frozen status for individual processes and messages considering, for example, the difficulty to trace them during debugging, achieving thus a desired transparency/performance trade-off.

Our scheduling algorithm will handle these transparency requirements by allocating the same start time[1] for v_i in all the alternative fault-tolerant schedules of application $\mathcal{A}$. For example, to handle the situation in Fig. 2d, where P_3 and its inputs m_2 and m_3 are not affected by faults, $\mathcal{T}(m_2)$, $\mathcal{T}(m_3)$ and $\mathcal{T}(P_3)$ will have to be set to "frozen".

5. Fault-Tolerant Conditional Process Graph

In Fig. 2 we have an application $\mathcal{A}$ modeled as a process graph $\mathcal{G}_1$, mapped on an architecture of two nodes, which can experience at most two transient faults. For scheduling purposes we will convert the application $\mathcal{A}$ to a *fault-tolerant conditional process graph* (FT-CPG) G. In an FT-CPG the fault occurrence information is represented as *conditional edges* and the frozen processes/messages are captured using *synchronization nodes*. The FT-CPG in Fig. 3 captures all the fault scenarios that can happen during the execution of application $\mathcal{A}$ in Fig. 2, considering the transparency requirements in Fig. 2d. For example, the subgraph marked with thicker edges and shaded nodes in Fig. 3 captures the worst-case schedule in Fig. 2d$_2$. The fault scenario for a given process execution, for example P_4^1, the first execution of P_4, is captured by the conditional edges $F_{P_4^1}$ (fault) and $\bar{F}_{P_4^1}$ (no-fault). The transparency requirement that, for example, P_3 has to be frozen, is captured by the synchronization node P_3^S.

Formally, an FT-CPG is a directed acyclic graph $G(V_P \cup V_C \cup V_T, E_S \cup E_C)$. Each node $P_i^j \in V_P$ is a regular node. A node $P_i^j \in V_C$ with *conditional edges* at the output is a *conditional process* that produces a condition. The condition value produced is "true" (denoted with $F_{P_i^j}$) if P_i^j experiences a fault, and "false" (denoted with $\bar{F}_{P_i^j}$) if P_i^j does not experience a fault. Alternative paths starting from such a process, which correspond to complementary values of the condition, are disjoint[2]. In Fig. 3, process P_1^1 is a conditional process because it "produces" condition $F_{P_1^1}$, while P_1^3 is a regular process. Each node $v_i \in V_T$ is a *synchronization node* and represents a frozen process or message (i.e., $\mathcal{T}(v_i)$ = *frozen*). In Fig. 3, m_2^S and P_3^S are synchronization nodes (depicted with a rectangle) representing a message and a process, respectively. Synchronization nodes take zero time to execute.

Regular and conditional processes are activated when all their inputs have arrived. However, a synchronization node can be activated (the process started or the message transmitted) after inputs coming on one of the alternative paths have arrived. For example, a transmission on the edge $\bar{F}_{P_1^1}$ will be enough to activate m_2^S. Moreover, a boolean expression K_{P_i}, called guard, can be associated to each node P_i in the graph. The guard captures the necessary activation conditions in a given fault scenario. In Fig. 3, for example, $K_{P_2^2} = \bar{F}_{P_1^1} \wedge F_{P_2^1}$ means that P_2^2 will be activated in the fault scenario where P_2 experienced a fault,

1. A frozen process P_i with a start time t_i, if affected by a fault, will be re-executed at a start time $t_i = C_i + \mu$.
2. They can only meet in a synchronization node.

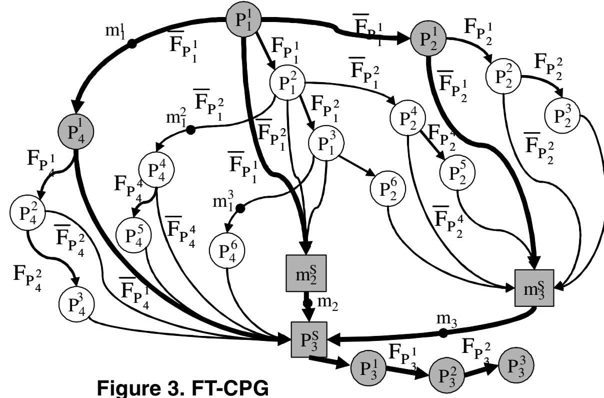


Figure 3. FT-CPG

while P_1 did not. A node is activated only when the value of the associated guard is true.

E_S and E_C are the sets of simple and conditional edges, respectively. An edge $e_{ij} \in E_S$ from P_i to P_j indicates that the output of P_i is the input of P_j. An edge $e_{ij} \in E_C$ is a *conditional edge* and has an associated condition value. Such an edge is P_1^1 to P_4^1 in Fig. 3, with the associated condition $\bar{F}_{P_1^1}$ as being "false". Transmission on conditional edges takes place only if the associated condition is satisfied.

Before applying the transformation of an application $\mathcal{A}$ to an FT-CPG G, we merge the application graphs $\mathcal{G}_i \in \mathcal{A}$ into one single merged graph $\mathcal{G}$, as detailed in [16], with a period equal to the LCM of all constituent graphs. In Fig. 3 we depict the FT-CPG G, which is the result of transforming the application $\mathcal{A}$ in Fig. 2, considering the transparency/trade-off requirements $\mathcal{T}(\mathcal{A})$ in Fig. 2d.

- Each process P_i is transformed into a structure which models the possible fault occurrence scenario in P_i, consisting of k conditional nodes and their corresponding conditional edges, and one regular node. For example, process P_4 from Fig. 2, which has to handle two transient faults, is transformed to conditional processes P_4^1 and P_4^2, conditional edges labelled $F_{P_4^1}$, $\bar{F}_{P_4^1}$, $F_{P_4^2}$ and $\bar{F}_{P_4^2}$, and regular process P_4^3. We denote with P_i^j the jth copy of $P_i \in \mathcal{A}$. In Fig. 3, P_4^1 is the first execution of P_4, P_4^2 is second execution of P_4, and P_4^3 is the last execution, which will not experience a fault, since $k = 2$.
- Each frozen process $P_i \in \mathcal{T}(\mathcal{A})$ or frozen message $m_i \in \mathcal{T}(\mathcal{A})$ is transformed into a synchronization node. For example, frozen message m_2 from Fig. 2 is transformed to the synchronization node m_2^S in Fig. 3.
- Each edge e_{ij} with its regular message m_i is copied into the new FT-CPG, into as many places as necessary, to connect the structures resulted from the transformations in the first two steps (see Fig. 3).

6. Scheduling FT-CPGs

The problem that we are addressing in this paper can be formulated as follows. Given an application $\mathcal{A}$, mapped on an architecture consisting of a set of hardware nodes $\mathcal{N}$ interconnected via a broadcast bus B, and a set of transparency requirements on the application $\mathcal{T}(\mathcal{A})$, we are interested to determine the schedule table $\mathcal{S}$ such that the worst-case end-to-end delay δ_G, by which the application completes execution is minimized, and the transparency requirements captured by $\mathcal{T}$ are satisfied. If the resulting delay is smaller than the deadline, the system is schedulable.

6.1 Schedule Table

The output produced by the FT-CPG scheduling algorithm is a schedule table that contains all the information needed for a distributed run time scheduler to take decisions on activation of processes. It is considered that, during execution, a very simple non-preemptive scheduler located in each node decides on process and communication activation depending on the actual values of conditions.

Only one part of the table has to be stored in each node, namely, the part concerning decisions that are taken by the corresponding scheduler. Fig. 4 presents the schedules for the nodes N_1 and N_2 produced by our scheduling algorithm in Fig. 5 for the FT-CPG in Fig. 3. In each table there is one row for each process and message from application $\mathcal{A}$. A row contains activation times corresponding to different values of conditions. In addition, there is one row for each condition whose value has to be broadcasted to other processors. Each column in the table is headed by a logical expression constructed as a conjunction of condition values. Activation times in a given column represent starting times of the processes and transmission of messages when the respective expression is true.

According to the schedule for node N_1 in Fig. 4, process P_1 is activated unconditionally at the time 0, given in the first column of the table. Activation of the rest of the processes, in a certain execution cycle, depends on the values of the conditions, i.e., the unpredictable occurrence of faults during the execution of certain processes. For example, process P_2 has to be activated at $t = 30$ if $\bar{F}_{P_1^1}$ is true, at $t = 100$ if $F_{P_1^1} \wedge F_{P_1^2}$ is true, etc. At a certain moment during the execution, when the values of some conditions are already known, they have to be used to take the best possible decisions on process activations. Therefore, after the termination of a process that produces a condition, the value of the condition is broadcasted from the corresponding processor to all other processors. This broadcast is scheduled as soon as possible on the communication channel, and is considered together with the scheduling of the messages. The scheduler in a node knows from its schedule table when to expect a condition message.

To produce a deterministic behavior, which is globally consistent for any combination of conditions (faults), the table has to fulfill several requirements:

1. No process will be activated if, for a given execution cycle, the conditions required for its activation are not fulfilled.
2. Activation times have to be uniquely determined by the conditions.
3. Activation of a process P_i at a certain time t has to depend only on condition values which are determined at the respective moment t and are known to the processing element which executes P_i.

N_1	*true*	$F_{P_1^1}$	$\bar{F}_{P_1^1}$	$F_{P_1^1} \wedge F_{P_1^2}$	$F_{P_1^1} \wedge \bar{F}_{P_1^2}$	$F_{P_1^1} \wedge \bar{F}_{P_1^2} \wedge F_{P_2^4}$	$F_{P_1^1} \wedge \bar{F}_{P_1^2} \wedge \bar{F}_{P_2^4}$	$\bar{F}_{P_1^1} \wedge F_{P_2^1}$	$\bar{F}_{P_1^1} \wedge F_{P_2^1} \wedge F_{P_2^2}$	$\bar{F}_{P_1^1} \wedge F_{P_2^1} \wedge \bar{F}_{P_2^2}$	$\bar{F}_{P_1^1} \wedge \bar{F}_{P_2^1}$
P_1	0 (P_1^1)	35 (P_1^2)		70 (P_1^3)							
P_2			30 (P_2^1)	100 (P_2^6)	65 (P_2^4)	90 (P_2^5)		55 (P_2^2)	80 (P_2^3)		
m_1			31 (m_1^1)	100 (m_1^3)	66 (m_1^2)						
m_2			105	105	105						
m_3				120		120	120		120	120	120
$F_{P_1^1}$	30										
$F_{P_1^2}$		65									

N_2	*true*	$F_{P_1^1}$	$\bar{F}_{P_1^1}$	$F_{P_1^1} \wedge F_{P_1^2}$	$F_{P_1^1} \wedge \bar{F}_{P_1^2}$	$F_{P_1^1} \wedge \bar{F}_{P_1^2} \wedge F_{P_4^1}$	$F_{P_1^1} \wedge \bar{F}_{P_1^2} \wedge \bar{F}_{P_4^1}$	$\bar{F}_{P_1^1} \wedge F_{P_4^1}$	$\bar{F}_{P_1^1} \wedge F_{P_4^1} \wedge F_{P_4^2}$	$\bar{F}_{P_1^1} \wedge F_{P_4^1} \wedge \bar{F}_{P_4^2}$	$\bar{F}_{P_1^1} \wedge \bar{F}_{P_4^1}$	$F_{P_3^1}$	$F_{P_3^1} \wedge F_{P_3^2}$
P_3				136 (P_3^8)		136 (P_3^1)	136 (P_3^1)		136 (P_3^1)	136 (P_3^1)	136 (P_3^1)	161 (P_3^2)	186 (P_3^3)
P_4			36 (P_4^1)	105 (P_4^6)	71 (P_4^4)	106 (P_4^5)		71 (P_4^2)	106 (P_4^3)				

Figure 4. Local Schedule Tables

6.2 Scheduling Algorithm

According to our application model, some processes can only be activated if certain conditions (i.e., fault occurrences), produced by previously executed processes, are fulfilled. Such process scheduling is complicated since at a given activation of the system, only a certain subset of the total amount of processes is executed and this subset differs from one activation to the other. As the values of the conditions are unpredictable, the decision on which process to activate and at which time has to be taken without knowing which values the conditions will later get. On the other side, at a certain moment during execution, when the values of some conditions are already known, they have to be used in order to take the best possible decisions on when and which process to activate, in order to reduce the schedule length.

Optimal scheduling has been proven to be an NP-complete problem [17] in even simpler contexts. Hence, heuristic algorithms have to be developed to produce a schedule of the processes such that the worst case delay is as small as possible. Our strategy for the synthesis of fault-tolerant schedules is presented in Fig. 5. The FTScheduleSynthesis function takes as input the application $\mathcal{A}$ with the transparency requirements $\mathcal{T}$, the number k of transient faults that have to be tolerated, the architecture consisting of processors $\mathcal{N}$ and bus B, the mapping $\mathcal{M}$, and produces the schedule table $\mathcal{S}$.

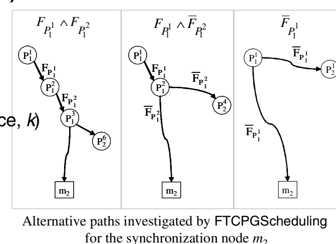
Our synthesis approach employs a *list scheduling* based heuristic, FTCPGScheduling, presented in Fig. 6, for scheduling each alternative fault-scenario. However, the fault scenarios cannot be independently scheduled: the derived schedule table has to fulfill the requirements (1) to (3) presented in Section 6.2, and the synchronization nodes have to be scheduled at the same start time in all alternative schedules.

In the first line of the FTScheduleSynthesis algorithm, we initialize the schedule table $\mathcal{S}$ and build the FT-CPG G as presented in Section 5. List scheduling heuristics use priority lists from which ready processes are extracted in order to be scheduled at certain moments. A process is ready if all its predecessors have been scheduled. We use the *partial critical path* (PCP) priority function [7] for ordering the ready list (line 3).

The property of a synchronization node S_i is that, in order to mask fault occurrences, it must have the same start time t_i in the schedule $\mathcal{S}$, regardless of the guard K_{S_i} under which is scheduled. For example, the synchronization node m_2 has the same start time of 105, in each corresponding column of the table. In order to determine the start time t_i of a synchronization node $S_i \in \mathcal{L}_S$, where $\mathcal{L}_S$ is the list of synchronization nodes, we will have to investigate all the alternative fault-scenarios (modeled as different alternative paths through the FT-CPG) that lead to S_i. Fig. 5, depicts the three alternative paths that lead to m_2 for the graph in Fig. 3. These paths are generated using the FTCPGScheduling function (called in line 6), which records the maximum start time t_{max} of S_i over the start times in all the alternative paths. In addition, FTCPGScheduling also records the guards $\mathcal{K}_{S_i}$ under which S_i has to be scheduled. The synchronization node S_i is then inserted into the schedule table in the columns corresponding to the guards in the set $\mathcal{K}_{S_i}$ at the unique time t_{max} (line 10 in Fig. 5). For example, m_2 is inserted at time $t_{max} = 105$ in the columns corresponding to $\mathcal{K}_{m_2} = \{ \overline{F}_{P_1^1}, F_{P_1^1} \wedge F_{P_1^2}, F_{P_1^1} \wedge \overline{F}_{P_1^2} \}$.

The FTCPGScheduling function is recursive and calls itself for each conditional node in order to separately schedule the nodes in the faulty branch, and those in the true branch (lines 21 and 23, Fig. 6). Thus, the

```
FTScheduleSynthesis(A, T, k, N, B, M)
1  S = ∅; G = BuildFTCPG(A, k)
2  L_S = GetSynchronizationNodes(G)
3  PCPPriorityFunction(G, L_S)
4  for each S_i ∈ L_S do
5     t_max = 0; K_Si = ∅
6     FTCPGScheduling(0, G, S_i, source, k)
7     for each K_j ∈ K_Si do
8        Insert(S, S_i, t_max, K_j)
9     end for
10 end for
11 return S
end FTScheduleSynthesis
```



Alternative paths investigated by FTCPGScheduling for the synchronization node m_2

Figure 5. Fault-Tolerant Schedule Synthesis Strategy

```
FTCPGScheduling(t, G, S, L, f)
1  for each R ∈ N ∪ {B} do
2   L_R = LocalReadyList(L, R)
3   while L_R ≠ ∅ do
4    P_i := Head(L_R)
5    t = ResourceAvailable(R, t) -- the earliest time when R is free
6    K = KnownConditions(R, t) -- the conditions known to R at time t
7    if P_i = S then -- synchronization node currently under investigation
8       if t > t_max then
9          t_max = t -- the latest start time is recorded
10         K_Si = K_Si ∪ {K} -- the guard of the synchronization node is recorded
11      end if
12      return -- exploration stops at the synchronization node S
13   else if P_i ∈ V_T and P_i unscheduled then -- other synchronization nodes
14      return -- are not scheduled at the moment
15   end if P_i ∈ E_C and BroadcastCondition(P_i) = false then
16      return -- the condition does not have to be broadcast to other processors
17   end if
18   Insert(S, P_i, t, K) -- the node is placed in the schedule
19   if P_i ∈ V_C and f > 0 then -- conditional process and faults can still occur
20      -- schedule the faulty branch
21      FTCPGScheduling(t, G, L ∪ GetReadyNodes(P_i, true), f - 1)
22      -- schedule the non-faulty branch
23      FTCPGScheduling(t, G, L ∪ GetReadyNodes(P_i, false), f)
24   else
25      L = L ∪ GetReadyNodes(P_i)
26   end if
27 end for
end FTCPGScheduling
```

Figure 6. FT-CPG Scheduling Algorithm

alternative paths are not activated simultaneously and resource sharing is correctly achieved. During the exploration of the FT-CPG it is important to eliminate alternative paths that are not possible to occur. This requirement is handled by introducing the parameter f, which represents the number of faults that still can occur. f is decremented for each call of FTCPGScheduling that explores a faulty (true) branch. Thus, only if $f > 0$ (line 19), we will continue to investigate branches through recursions.

For each resource R, the highest priority node is removed from the head of the local priority list $\mathcal{L}_R$ (line 2). If the node is the currently investigated synchronization node S, the largest start time and the current guards are recorded (lines 9–10). If other unscheduled synchronization nodes are encountered, they will not be scheduled yet (lines 13–14), since FTCPGScheduling investigates one synchronization node at a time. Otherwise, the current node P_i is placed in the schedule $\mathcal{S}$ at time t under guards K. The time t is the time when the resource R is available. Our definition of resource availability is different from classical list scheduling. Since we enforce the synchronization nodes to start at their latest time t_{max} to accommodate all the alternative paths, we might have to insert idle time on the resources on those alternative paths that finish sooner than t_{max}. Thus, our ResourceAvailable function will determine the first contiguous segment of time which is available on R, large enough to accommodate P_i. For example, m_2 is scheduled first at 105 on the bus, thus time 0–105 is idle time on the bus. We will later schedule m_1 at times 66, 31 and 100, within this idle segment. The scheduling of P_i will be done under the currently known conditions K, determined at line 6 on the resource R. Our approach eliminates from K those conditions that although known to R at time t, will not influence the execution of P_i.

For efficiency reasons, our implementation will use as input the non fault-tolerant graph structure $\mathcal{A}$ presented in Section 3. Starting from $\mathcal{A}$, the GetReadyNodes functions (lines 21, 23 and 25) will insert nodes into the ready lists as if our algorithm would visit the FT-CPG G. Moreover, we have limited the levels of recursion to at most k.

7. Experimental Results

For the evaluation of our algorithms we used applications of 20, 40, 60, and 80 processes mapped on architectures consisting of 4 nodes. We have varied the number of faults, considering 1, 2, and 3 faults, which can happen during one execution cycle. The duration µ of the recovery time has been set to 5 ms. Fifteen examples were randomly generated for each application dimension, thus a total of 60 applications were used for experimental evaluation. Execution times and message lengths were assigned randomly 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 how well the proposed scheduling algorithm handles the transparency/performance trade-offs imposed by the designer. Hence, we have scheduled each application, on its corresponding architecture, using the FTScheduleSyntheis (FTSS) strategy from Fig. 5. In order to evaluate FTSS, we have derived a reference non-fault tolerant implementation, NFT. The NFT approach uses a list-scheduling strategy to build a non-fault tolerant schedule. To the NFT implementation obtained, we would like to add fault-tolerance with as little as possible overhead, without adding any extra hardware resources. Let δ_{FTSS} and δ_{NFT} be the end-to-end delays of the application obtained using FTSS and NFT, respectively. The overhead is defined as $100 \times (\delta_{FTSS} - \delta_{NFT}) / \delta_{NFT}$.

For the experiments, we considered that the designer is interested to maximize the amount of transparency for the inter-processor messages, which are critical to a distributed fault-tolerant system. Thus, we have considered five transparency scenarios, depending on how many of the inter-processor messages have been set as transparent: 0, 25, 50, 75 or 100%. Table 1 presents the average fault-tolerance overheads for each of the five transparency requirements. We can see that as the transparency requirements are relaxed, our scheduling approach is able to improve the performance of the application, producing good quality results in terms of fault-tolerance overheads. For example, for application graphs of 60 processes with three faults, we have obtained an 86% overhead for 100% transparency, which was reduced to 58% for 50% transparency.

Table 2 presents the average memory[1] per processor (in kilobytes) required by the schedule tables. Often, the an entity has the same start time under different conditions. Such entries into the table can be merged into a single table entry, headed by the union of the logical expressions.Thus, Table 2 reports the memory required after such a straightforward compression. We can observe that as the transparency increases, the memory requirements decrease. For example, for 60 processes and three faults, increasing the transparency from 50% to 100% reduces the memory needed from 18K to 4K.

The FTSS algorithm runs in less than three seconds for large applications (80 processes) when only one fault has to be tolerated. Due to the nature of the problem, the execution time increases exponentially with the number of faults that have to be handled. However, even for graphs of 60 processes, for example, and three faults, the schedule synthesis algorithm finishes in under 10 minutes.

The approach presented by us in [9] can only handle a setup with 100% transparency for inter-processor messages. As a second set of experiments, we have compared the FTSS approach with the list scheduling based approach, namely LS, proposed in [9], considering this 100% scenario. Besides the fact that FTSS supports complete flexibility regarding the degree of transparency, there is a second important difference between FTSS and LS. Due to the FT-CPG scheduling approach, FTSS will generate the best possible schedule for each fault scenario, where both execution order and start time of processes are adapted to the actual situation. LS, however, only adapts the start time, but does not change the execution order across fault scenarios. In order to compare the two algorithms, we have produced the end-to-end delay δ_{LS} of the application when using LS. When comparing the delay δ_{FTSS} obtained by our approach with δ_{LS} for the experiments in the case of k = 2, for example, our approach outperforms LS on average with 13, 11, 17, and 12% for application dimensions of 20, 40, 60 and 80 processes, respectively.

Table 1. Fault-Tolerance Overheads

$\lvert\mathcal{V}\rvert$	20			40			60			80		
	k=1	k=2	k=3	k=1	k=2	k=3	k=1	k=2	k=3	k=1	k=2	k=3
100%	48	86	139	39	66	97	32	58	86	27	43	73
75%	48	83	133	34	60	90	28	54	79	24	41	66
50%	39	74	115	28	49	72	19	39	58	14	27	39
25%	32	60	92	20	40	58	13	30	43	10	18	29
0%	24	44	63	17	29	43	12	24	34	8	16	22

Table 2. Memory Requirements

$\lvert\mathcal{V}\rvert$	20			40			60			80		
	k=1	k=2	k=3	k=1	k=2	k=3	k=1	k=2	k=3	k=1	k=2	k=3
100%	0.13	0.28	0.54	0.36	0.89	1.73	0.71	2.09	4.35	1.18	4.21	8.75
75%	0.22	0.57	1.37	0.62	2.06	4.96	1.20	4.64	11.55	2.01	8.40	21.11
50%	0.28	0.82	1.94	0.82	3.11	8.09	1.53	7.09	18.28	2.59	12.21	34.46
25%	0.34	1.17	2.95	1.03	4.34	12.56	1.92	10.00	28.31	3.05	17.30	51.30
0%	0.39	1.42	3.74	1.17	5.61	16.72	2.16	11.72	34.62	3.41	19.28	61.85

1. Considering an architecture where an *int* and a *pointer* are represented on two bytes.

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 [16]. 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 300 ms, $k = 2$ and $\mu = 2$ ms.

Considering 100% transparency for the messages on the bus, LS produced an end-to-end delay of 384, larger than the deadline. Our FTSS approach reduced this delay to 346 given the 100% transparency, which is still unschedulable. If we relax this transparency requirement and select only half of the messages as transparent, we are able to further reduce the delay to 274 which meets the deadline. The designer can use our scheduling synthesis approach to explore several design alternatives to find that one which provides the most useful transparency properties. For example, the CC is still schedulable even with 70% transparency.

8. Conclusions

In this paper we have proposed a novel scheduling approach for fault-tolerant embedded systems in the presence of multiple transient faults. Both processes and messages are statically scheduled, and we have considered process re-execution for tolerating faults. The main contribution of our schedule synthesis approach is the ability to handle the performance versus transparency and memory trade-offs imposed by the designer. The algorithm uses fault-occurrence information to improve the schedule generation. Thus, we are able 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] V. Claeson, S. Poldena, J. Söderberg, “The XBW Model for Dependable Real-Time Systems”, *Parallel and Distributed Systems Conf.*, 1998.

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

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

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

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

[8] C. C. Han, K. G. Shin, 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 et al., “Design Optimization of Time- and Cost-Constrained Fault-Tolerant Distributed Embedded Systems”, *DATE Conf.*, 864-869*,* 2005.

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

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

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

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

[14] H. Kopetz, Günter Bauer, “The Time-Triggered Architecture”, *Proc. of the IEEE*, 91(1), 112–126, 2003.

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

[16] 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.

[17] D. Ullman, “NP-Complete Scheduling Problems,” in *J. of Computer Systems Science*, vol. 10, 384–393, 1975.

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