Mapping of Fault-Tolerant Applications with Transparency on Distributed Embedded Systems

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

Published in:
Euromicro Conference on Digital System Design

Link to article, DOI:
10.1109/DSD.2006.65

Publication date:
2006

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

Mapping of Fault-Tolerant Applications with Transparency on 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 for the mapping optimization of fault-tolerant embedded systems for safety-critical applications. Processes and messages are statically scheduled. Process re-execution is used for recovering from multiple transient faults. We call process recovery transparent if it does not affect operation of other processes. Transparent recovery has the advantage of fault containment, improved debugability and less memory needed to store the fault-tolerant schedules. However, it will introduce additional delays that can lead to violations of the timing constraints of the application. We propose an algorithm for the mapping of fault-tolerant applications with transparency. The algorithm decides a mapping of processes on computation nodes such that the application is schedulable and the transparency properties imposed by the designer are satisfied. The mapping algorithm is driven by a heuristic that is able to estimate the worst-case schedule length and indicate whether a certain mapping alternative is schedulable.

1. Introduction

Transient faults are becoming the most common types of faults in modern electronic components due to raising levels of integration in semiconductors, lower voltage levels, and higher operational frequency. Causes of transient faults are electromagnetic interference, cosmic radiation, power supply fluctuations, and many others.

Transient faults are often discriminated from intermittent faults, or internal system disturbances [2], which could result from electromagnetic interference of internal components, crosstalk, or overheating. The approaches presented in this paper are suitable for tolerating both transient and intermittent faults. Although transient and intermittent faults do not lead to a permanent damage of the circuit, errors caused by them can be fatal for many safety-critical applications, such as those used in avionics, automotive, factory automation and medical systems.

Researchers have proposed several hardware architecture solutions, such as MARS [15], TTA [16] 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 a 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, and 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 priority based scheduling [1, 3, 9, 22]. Considering their high degree of predictability, researchers have also proposed approaches for integrating fault-tolerance into the framework of static scheduling [14]. A simple heuristic for combining several static schedules in order to mask fault-patterns through replication is proposed in [5], without, however, considering any timing constraints. This approach is used as the basis for cost and fault-tolerance trade-offs within the Metropolis environment [17].

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 [12] 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 [13]. In [11], we have proposed a fine-grained approach to scheduling with 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 have proposed a conditional scheduling algorithm for the synthesis of fault tolerant schedules that can handle the transparency/performance trade-offs imposed by the designer, without introducing unnecessary delays.

*** This work was partially supported by the National Graduate School in Computer Science (CUGS) of Sweden.**

In [10] 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. Design optimization was based on a heuristic and contained three stages, scheduling, mapping and fault tolerance policy assignment. The algorithms were iterating until a first schedulable fault-tolerant solution was found. However, the approach in [10] was incapable of handling transparency properties customized by the designer and could not be used for trading-off transparency for performance. Therefore, in this work we focus on mapping optimization of fault-tolerant embedded systems such that the application is not only schedulable but also transparency properties imposed by the designer are satisfied.

Experiments have shown that due to its long runtime the conditional scheduling algorithm from [11] cannot be used inside an optimization loop. Therefore, in this paper, we propose a fast scheduling heuristic, which is able to estimate the worst-case schedule length, thus guiding an iterative mapping optimization. The heuristic can accurately indicate whether a certain mapping alternative is schedulable.

The next two sections present the system architecture and the application model, respectively. Section 4 discusses the importance of considering transparency properties during mapping optimization. Section 5 proposes a mapping optimization strategy and mapping algorithm. Section 6 presents our schedule length estimation heuristic. 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 [14]. 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 [12] 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) [20]. 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.

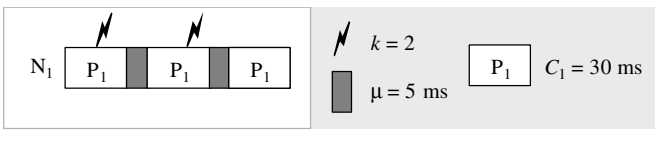

Figure 1. Re-execution

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 $\mu = 5$ 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 their execution.

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.

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 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.

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). For the purposes of mapping and scheduling we use a merged application graph $\mathcal{G}$, obtained by merging all the graphs in the application, and which has a period equal to LCM of the periods of all constituent graphs [18].

4. Mapping with Transparency

Transparent recovery has the advantages of fault containment, improved debugability and less memory needed to store the

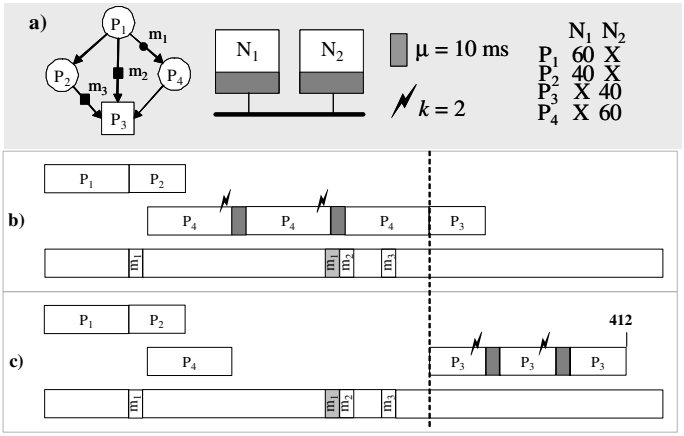

Figure 2. Application with Transparency

fault-tolerant schedules. A designer would like to introduce as much transparency as possible. However, transparency will introduce delays that can violate the timing constraints of the application.

We consider a fine-grained approach to transparency as in [11]. The designer specifies desired degree of transparency by declaring certain processes and messages as *frozen*. A frozen process or message has a fixed start time regardless of the occurrence of faults in the rest of the application (frozen processes and messages are depicted using squares). The debugability of the application is improved by transparency because it is easier to observe the behavior of frozen processes and messages in alternative schedules that correspond to different fault scenarios.

Let us illustrate the timing overhead introduced with transparency. Fig. 2 depicts two alternative schedules of the

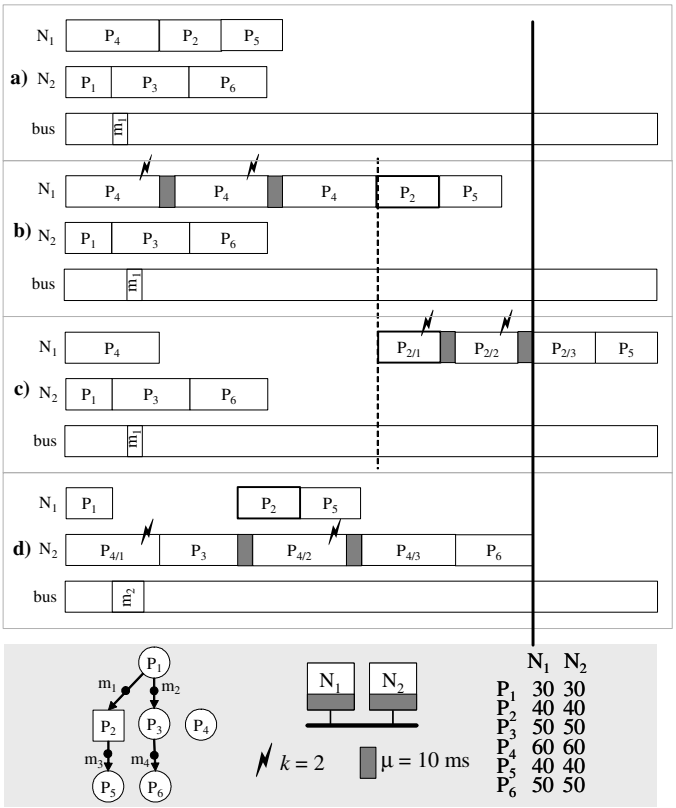

Figure 3. Mapping and Frozen Processes

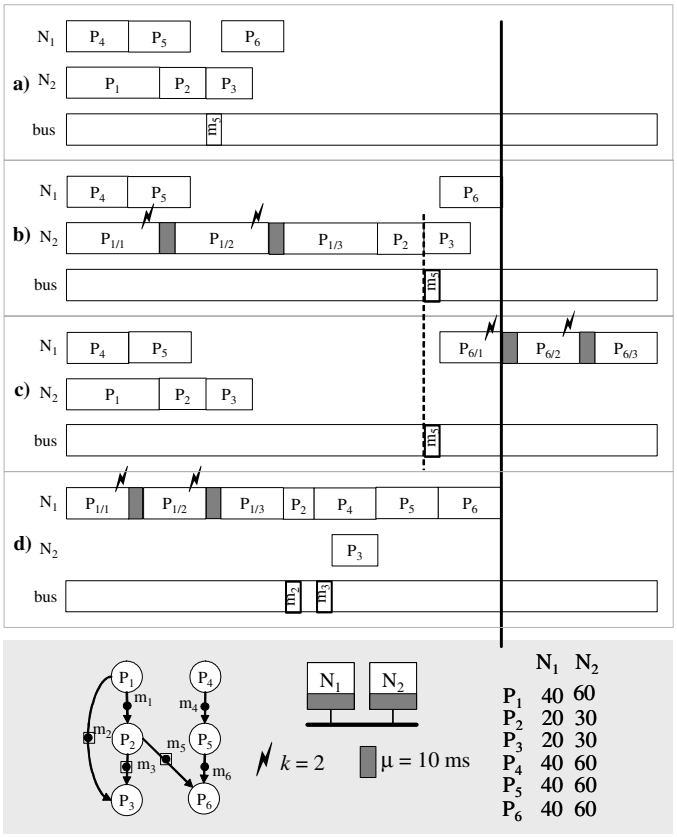

Figure 4. Mapping and Frozen Messages

application composed of four processes and three messages, where process P_3 and messages m_2 and m_3 are frozen. Since P_3, m_2 and m_3 are frozen they should be scheduled at one single time independently of external fault occurrences. For example, re-executions of process P_4 in case of faults on Fig. 2b must not affect the start time of process P_3. As a result, even if no faults happen in process P_4, as illustrated on Fig. 2c, process P_3 will have to be delayed. Process P_3 experiences two faults and is re-executed. Due to transparency, the idle time between processes P_4 and P_3 cannot be utilized, which leads to a long schedule of 412 ms. Similarly, idle times are introduced before messages m_2 and m_3, such that re-executions of processes P_1 and P_2 do not affect the sending times of these messages. Contrary, message m_1 can be sent at different times. In both alternative schedules message m_1 is sent immediately after process P_1. m_1 will be delayed only if process P_1 experiences faults (as shown with the grey rectangle labelled "m_1" on the bus).

Transparency properties of applications have to be taken into account during design optimization. In our application model, given an application $\mathcal{A}(\mathcal{V}, \mathcal{E})$ we will capture the transparency using the function $\mathcal{T}: \mathcal{V} \rightarrow \mathit{frozen}$, where $v_i \in \mathcal{V}$ is a node in the application graph, which can be either a process or a communication message.

4.1 Motivational Examples

In this work we are interested to find a mapping of processes such that the delays introduced due to the frozen processes are reduced. To illustrate the issues related to mapping with transparency, we will discuss two examples, one with a frozen process, presented in Fig. 3 and one with frozen messages, presented in Fig. 4.

In Fig. 3 we consider an application consisting of six processes, P_1 to P_6 that have to be mapped on an architecture consisting of two computation nodes connected to a bus. We assume that there can be at most $k = 2$ faults during one cycle of operation. The worst-case execution times for each process on each computation node are depicted in the table next to the architecture. Furthermore, let us assume that process P_2 is frozen (depicted with a rectangle in the application graph in Fig. 3). We impose a deadline of 300 ms for the application (a thick line crossing the figure). If we decide the mapping without considering the transparency requirement on P_2, we obtain the optimal mapping depicted in Fig. 3a (processes P_2, P_4 and P_5 are mapped on node N_1; P_1, P_3 and P_6 are mapped on node N_2). If transparency is ignored, the application is schedulable in all possible fault scenarios; it meets the deadline even in case of the worst-case fault scenario shown in Fig. 3b.

If the same mapping determined in Fig. 3a is used in the case of transparency, obtained solution is depicted in Fig. 3c. However, in this case, the deadline will not be met due to the delay introduced by the frozen process P_2. A mapping that makes the system schedulable even in the worst-case fault scenario and with a frozen P_2 is shown in Fig. 3d. According to this mapping, processes P_1, P_2, P_4, P_5 and P_6 are mapped on node N_1 and only process P_3 is mapped on node N_2. Counterintuitively, this mapping is not balanced and communications are increased compared to the previous mapping, since we send message m_2, which is two times larger than message m_1.

Another example, illustrating the importance of considering frozen messages during the mapping process, is depicted in Fig. 4. The application consists of six processes, P_1 to P_6. Let us consider that all messages are frozen. This means that messages will have the same start time on the bus regardless of the fault scenario. The optimal mapping, ignoring transparency, is presented in Fig. 4a. The application meets the deadline even in the worst case fault scenario as shown in Fig. 4b. This mapping is balanced and communications are minimized. Once we introduce transparency, the application becomes unschedulable, as illustrated in Fig. 4c. However, considering the transparency requirements during scheduling leads to a schedulable solution depicted in Fig. 4d. Again, counterintuitively, the solution in Fig. 4d is not balanced and instead of one message two messages are sent via the bus.

The examples presented have shown that transparency properties have to be carefully considered during mapping and

```
OptimizationStrategy(G, k, N, T, D)
1  M_init := InitialMapping(G)
2  l := CondScheduling(G, k, N, M_init, T)
3  if l < D then return M_init
4  while not_termination do
5     M_new := IterativeMapping(G, k, N, M_init, T)
6     l := CondScheduling(G, k, N, M_new, T)
7     if l < D then return M_new
8     M_init := FindNewInit(G, M_new)
9  end while
10 return no_solution
end OptimizationStrategy
```

Figure 5. Optimization Strategy

	true	F_{P_1}	$\overline{F}_{P_1}$	$F_{P_1} \wedge F_{P_1}$	$F_{P_1} \wedge \overline{F}_{P_1}$	$F_{P_1} \wedge \overline{F}_{P_1} \wedge F_{P_2}$
P_1	0	35		70		
P_2			30	100	65	90
m_1			31	100	66	
m_2			105	105	105	
m_3				120		120

Figure 6. Conditional Schedule Table

that mapping solutions which are optimal for non-transparent solution are inefficient when transparency is introduced.

5. Optimization Strategy

Our mapping optimization strategy, outlined in Fig. 5, determines a mapping $\mathcal{M}$ for application $\mathcal{A}$ on computation nodes $\mathcal{N}$ such that the application is schedulable and the transparency requirements $\mathcal{T}$ are satisfied. The optimization strategy receives as input the application graph $\mathcal{G}$, the maximum number of faults k in the system period, the architecture $\mathcal{N}$, transparency requirements $\mathcal{T}$, and deadline D. The output of the algorithm is a mapping $\mathcal{M}$ of processes to nodes, and a conditional schedule table for processes and messages.

A conditional schedule table captures the alternative fault scenarios. Each column in the table corresponds to actual fault occurrences, and each row to a process or a message. A part of a conditional schedule table is depicted in Fig. 6. For example, process P_1 will always start at 0, e.g. at the beginning of each execution cycle. The notions F_{P_1} and $\overline{F}_{P_1}$ indicate the presence and absence of a fault in process P_1, respectively. For example, process P_2 will start at 30 if there is no fault in process P_1, at 100 in case of two faults in process P_1, and at 65 in case of one fault in P_1. Similarly, F_{P_2} indicates a fault in process P_2. For example, process P_2 will be re-executed at 90 if its first execution has failed.

The design problem outlined above is NP complete [21]. Our strategy, presented in Fig. 5, is to address separately the mapping and scheduling. We start by determining an initial mapping $\mathcal{M}_{init}$ with the InitialMapping function in line 1. This is a straightforward mapping that balances processor utilization and minimizes communications. The schedulability of the resulted system is evaluated with the conditional scheduling algorithm from [11] (lines 2-3). If the initial mapping is unschedulable, then we iteratively improve the mapping of processes on the critical path of the worst-case fault scenario aiming at finding a schedulable solution (lines 4-9). For this purpose, we use a hill-climbing heuristic, which combines a greedy algorithm and a method to recover from a local optimum.

A new mapping alternative $\mathcal{M}_{new}$ is obtained with a greedy algorithm, IterativeMapping, line 5, which is presented in the next section. Since IterativeMapping is a greedy heuristic it will very likely end up in a mapping $\mathcal{M}_{new}$, which is a local minimum. In order to explore other areas of the design space, we will restart the IterativeMapping heuristic with a new initial solution $\mathcal{M}_{init}$ which should not lead to the same local minimum. As recommended in the literature [19], an efficient way to find such a new initial mapping is the following. Given the actual solution $\mathcal{M}_{new}$ we apply an optimization run using a new cost function different from the global schedule length, which is used as a cost function in the optimization so far. This

```
IterativeMapping(G, k, N, M, T)
1  improvement := true
2  l_best := ScheduleLength(G, k, N, M, T)
3  while improvement do
4    improvement := false
5    P_best := ∅; N_best := ∅
6    CP := FindCP(G)
7    SortCP(CP)
8    for each P_i ∈ CP do
9      for each N_j ≠ N_c do
10       ChangeMapping(M, P_i, N_j)
11       l_new := ScheduleLengthEstimation(G, k, N, M, T)
12       RestoreMapping(M)
13       if l_new < l_best then
14         P_best := P_i; N_best := N_j; l_best := l_new
15         improvement := true
16       end if
17     end for
18   end for
19   if improvement then ChangeMapping(M, P_best, N_best)
20 end while
21 return M
end IterativeMapping
```

Figure 7. Iterative Mapping Heuristic (IMH)

optimization run will produce a new mapping $\mathcal{M}_{init}$ and is implemented by the function FindNewInit (line 8). This function runs a simple greedy iterative mapping, which is aiming at an optimal load balancing of the nodes.

If the solution produced by IterativeMapping is schedulable, then the optimization will stop (line 7). However, a termination criterion is needed in order to terminate the mapping optimization if no solution is found. A termination criterion, which we obtained empirically and which produced very good results, is to limit the number of iterations *without improvement* to $N_{proc} \cdot k \cdot \ln(N_{nodes})$, where N_{proc} is the number of processes, N_{nodes} is the number of computation nodes, and k is the maximum number of faults in the system period.

5.1 Iterative Mapping

IterativeMapping depicted in Fig. 7 is a greedy algorithm that incrementally changes the mapping $\mathcal{M}$ until no further improvement (line 3) is possible. Our approach is to tentatively change the mapping of processes on the critical path of the application graph $\mathcal{G}$. The critical path $\mathcal{CP}$ is found by the function FindCP (line 6). Each process $P_i \in \mathcal{CP}$ on the critical path is tentatively moved to each node in $\mathcal{N}$. We evaluate each move in terms of schedule length, considering transparency properties $\mathcal{T}$ and the number of faults k (line 11). The calculation of the schedule length should, in principle, be performed by the scheduling function ConditionalScheduling, presented by us in [11]. However, the scheduling takes too long time to be used inside such an iterative optimization loop. Therefore, we have developed a fast schedule length estimation heuristic, ScheduleLengthEstimation, which is used to guide the InitialMapping heuristic. The estimation heuristic is presented in Section 6.2.

After evaluating possible alternatives, the best move composed of the best process P_{best} and the best computation node N_{best} is selected (lines 13-16). This move is conserved if leading to improvement (line 19). IterativeMapping will stop if there is no further improvement.

6. Scheduling Heuristics

The conditional schedule table introduced in Section 5 is produced by the ConditionalScheduling algorithm from [11]. The ConditionalScheduling algorithm uses an internal representation of the possible fault scenarios, captured by a fault-tolerant conditional process graph presented in the next section.

The ConditionalScheduling algorithm traces all alternative scheduling paths corresponding to fault scenarios. The number of alternative paths to investigate is growing exponentially with the number of processes and, especially, with the number of faults. Hence, the execution time of the ConditionalScheduling algorithm is also growing exponentially as our experiments in Section 7 show. In Section 6.2 we propose a fast schedule length estimation heuristic, which is also based on the fault-tolerant conditional process graph representation, that is able to approximate the schedule length without generating the conditional schedule table.

6.1 Fault-Tolerant Conditional Process Graph

In Fig. 8 we have an application $\mathcal{A}$ modeled as a process graph $\mathcal{G}$, mapped on an architecture of two nodes, which can experience at most two transient faults. For scheduling purposes, we convert the original application graph $\mathcal{G}$ into a *fault-tolerant conditional process graph* (FT-CPG). 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. 8b captures all the fault scenarios that can happen during the execution of application $\mathcal{A}$ in Fig. 8a, considering the transparency requirements, depicted as rectangles on the process graph $\mathcal{G}$. For example, the subgraph marked with thicker edges and shaded nodes in Fig. 8b captures the

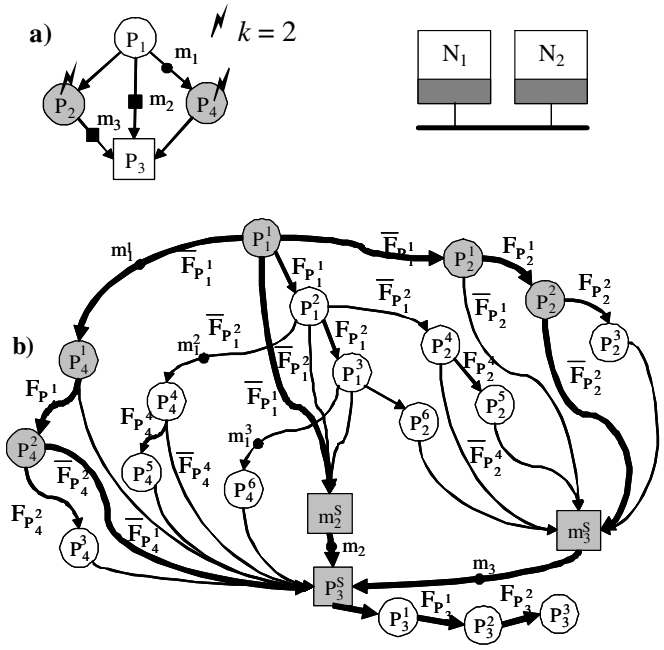

Figure 8. FT-CPG

alternative schedule corresponding to the fault scenario where one fault happens in process P_2 and one fault happens in process P_4. 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.

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[1]. In Fig. 8b, 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. 8b, 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.

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. 8b, with the associated condition $\bar{F}_{P_1^1}$. Transmission on conditional edges takes place only if the associated condition is satisfied.

In Fig. 8b we depict the FT-CPG G, which is the result of transforming the application $\mathcal{A}$ in Fig. 8a, considering the transparency/trade-off requirements $\mathcal{T}(\mathcal{A})$, depicted as rectangles on the process graph $\mathcal{G}$ in Fig. 8a.

- 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. 8a, 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 j^{th} copy of $P_i \in \mathcal{A}$. In Fig. 8b, 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. 8a is transformed to the synchronization node m_2^S in Fig. 8b.
- 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. 8b).

1. They can only meet in a synchronization node.

6.2 Schedule Length Estimation

The worst-case fault scenario consists of a combination of k fault occurrences that leads to the longest schedule length. We have proposed a conditional scheduling algorithm [11] that examines all the fault scenarios captured by the FT-CPG, produces the conditional schedule table, and implicitly determines the worst-case fault scenario. However, the algorithm is too slow for mapping optimization. Hence, in this section, we are proposing a worst-case schedule length estimation heuristic.

The main idea of the ScheduleLengthEstimation algorithm is to avoid examining all the fault scenarios, which is time-consuming. Instead, the estimation heuristic incrementally builds a fault scenario, which is as close as possible (in terms of resulted schedule length) to the worst case.

Considering a fault scenario $X(m)$ where m faults have occurred, we construct the fault scenario $X(m+1)$ with $m+1$ faults in a greedy fashion. Each fault scenario $X(m)$ corresponds to a partial FT-CPG $G_{X(m)}$, which includes only paths corresponding to the m fault occurrences considered in $X(m)$. Thus, we investigate processes from $G_{X(m)}$ to determine the process $P_i \in G_{X(m)}$ that introduces the largest delay if it experiences the $m^{\text{th}}+1$ fault (and has to be re-executed). A fault occurrence in P_i is then considered as part of the fault-scenario $X(m+1)$, and the iterative process continues until we reach k faults.

In order to speed up the estimation, we do not investigate all the processes in $G_{X(m)}$. Instead, our heuristic selects processes whose re-executions will likely introduce the largest delay. Candidate processes are those which have a long worst-case execution time and those which are located on the critical path.

The ScheduleLengthEstimation heuristic is outlined in Fig. 9. The set of all synchronization nodes $\mathcal{L}_s$ is generated (line 1). Priorities are assigned to all synchronization nodes (line 2). For priority assignment we use the partial critical path function outlined in [6]. The estimation chooses synchronization nodes according to the assigned priorities such that it can derive their fixed start time. For each synchronization node

```
ScheduleLengthEstimation(G, T, k, N, M)
1  L_S = GetSynchronizationNodes(G)
2  PCPPriorityFunction(G, L_S)
3  X(0) := ∅ ; ψ := SinkNode(G)
4  for each S_i ∈ L_S and ψ do
5    t_max := 0
6    Z := SelectProcesses(S_i, G)
7    for m := 1...k do
8      for each P_i ∈ Z do
9        G_X(m),i := CreatePartialFTCPG(X(m - 1), P_i)
10       t = ListScheduling(G_X(m),i , S_i)
11     end for
12     if t_max < t then
13       t_max := t ; P_worst := P_i
14     end if
15     X(m) := X(m - 1) + P_i
16   end for
17   Schedule(S_i, t_max)
18 end for
19 l_est := completion_time(ψ)
20 return l_est
end ScheduleLengthEstimation
```

Figure 9. Schedule Length Estimation

ScheduleLengthEstimation selects a set of processes that will potentially introduce the largest delay (line 6).

Re-executions of the selected processes are considered when the partial FT-CPG is generated (line 9). Fault scenarios are evaluated with a ListScheduling heuristic that stops once it reaches a synhronization node (line 10). The fault scenario that led to the greatest start time t_{max} is saved (line 15). Once the fault scenario of k faults $X(k)$ is obtained, the synchronization node is scheduled.

When all synchronization nodes are scheduled, the algorithm returns the worst-case schedule length.

7. Experimental Results

For evaluation of our mapping optimization strategy we used applications of 20, 30, and 40 processes (all unmapped), implemented on the architecture of 4 computation nodes. We have varied the number of faults from 2 to 4 within one execution cycle. The recovery overhead μ was set to 5 ms. Thirty examples were randomly generated for each dimension both with random structure and graphs based on more regular structure like trees and groups of chains. Execution times and message lengths were assigned randomly using uniform distribution within the 10 and 100 ms, and 1 to 4 bytes ranges, respectively. We have selected a transparency level with 25% frozen processes and 50% frozen messages. The experiments were done on Pentium 4 at 2.8GHz with 1 Gb of memory.

We were first interested to evaluate the proposed heuristic for schedule length estimation (ScheduleLengthEstimation in Fig. 9, denoted with SE) in terms of monotonicity relative to the ConditionalScheduling (CS) algorithm presented by us in [11]. SE is monotonous with respect to CS if for two alternative mapping solutions $\mathcal{M}_1$ and $\mathcal{M}_2$ it is true that if $CS(\mathcal{M}_1) \leq CS(\mathcal{M}_2)$ then also $SE(\mathcal{M}_1) \leq SE(\mathcal{M}_2)$.

For the purpose of evaluating the monotonicity of SE, 50 random mapping changes were performed for each application. Each of those mapping changes was evaluated with both SE and CS. The results are depicted in Table 1. As we can see, in over 90% of the cases, SE correctly evaluates mapping decisions, e.g. in the same way as CS. The rate of monotonicity decreases slightly with the application dimension. However, it is not influenced by increasing the number of faults.

Table 1. Monotonicity [%]

Procs.	2	3	4
20	94.20	90.58	91.65
30	89.54	88.90	91.48
40	88.91	86.93	86.32

Table 2. Execution Time [sec]

Procs.	2		3		4	
	SE	**CS**	**SE**	**CS**	**SE**	**CS**
20	0.01	0.07	0.02	0.28	0.04	1.37
30	0.13	0.39	0.19	2.93	0.26	31.50
40	0.32	1.34	0.50	17.02	0.69	318.88

Table 3. Mapping Improvement [%]

Procs.	2	3	4
20	32.89	32.20	30.56
30	35.62	31.68	30.58
40	28.88	28.11	28.03

Another important property of SE is its execution time, presented on Table 2. Execution time of the SE is growing linearly with the number of faults and application size. Over all graph dimensions, the execution time of SE is always less than 1 sec. In comparison, the execution time of CS is growing exponentially with the number of processes and the number of faults, see Table 2, and can reach 318.88 seconds for 40 processes and 4 faults. This shows that the conditional scheduling cannot be used inside the optimization loop, while the scheduling heuristic is well-suited for design space exploration.

We were also interested to evaluate our mapping optimization strategy. Table 3 shows the improvement by mapping optimization that considers fault tolerance with transparency over straightforward mapping, InitialMapping on Fig. 5, which does not consider the fault tolerance aspects. Thus, we determined using ConditionalScheduling the schedule length for two mapping alternatives: InitialMapping and the mapping obtained by our OptimizationStrategy in Fig. 5. Table 3 presents the percentage improvement in terms of schedule length of our mapping optimization compared to the straightforward solution. The schedule length obtained with our mapping optimization algorithm is 30% shorter on average. This shows that considering the fault tolerance and transparency aspects leads to significantly better design solutions and that the ES heuristic can be successfully used inside an optimization loop.

We were also interested to compare the solutions obtained using ES with the case where CS is used for evaluating the mapping alternatives. However, this comparison was possible only for applications of 20 processes. We chose 15 synthetic applications with 25% frozen processes and 50% frozen messages. In terms of schedule length, in case of 2 faults, the CS-based strategy was only 3.18% better than the ES-based. In case of 3 faults, the difference was 9.72%, while for 4 faults the difference in terms of obtained schedule length was of 8.94%.

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 300 ms, k = 2 and μ = 2 ms. The straightforward solution was unschedulable even with only 25% frozen messages and no frozen processes. However, the application optimized with our mapping strategy, was easily schedulable with 85% frozen messages. Moreover, we could additionally introduce 20% frozen processes without violating timing constraints.

8. Conclusions

In this paper we have addressed the mapping optimization of distributed embedded systems for fault-tolerant hard real-time applications. The processes and messages are scheduled with static cyclic scheduling. We have employed process re-execution as the fault-tolerance technique for tolerating transient faults.

Transparency has the advantages of fault containment, improved debugability and less memory needed to store the fault-tolerant schedules. The main contribution of our approach is the ability to consider the transparency requirements imposed by the designer during the mapping optimization.

The mapping is driven by a schedule estimation heuristic which is able to accurately evaluate a given mapping decision. Our experiments have shown that the schedule estimation heuristic is able to successfully guide the design space exploration.

Considering the fault-tolerance and transparency requirements during the mapping optimization process we are able to provide transparency-aware 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. Bondavalli et al., "Threshold-based Mechanisms to Discriminate Transient from Intermittent Faults", *IEEE Trans. on Computers*, 49(3), 230-245, 2000.

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

[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 Trans. 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 Trans. on Computers*, 52(3), 362–372, 2003.

[10] V. Izosimov et al., "Design Optimization of Time- and Cost-Constrained Fault-Tolerant Distributed Embedded Systems", *Proc. Design Automation and Test in Europe Conference*, 864-869, 2005.

[11] V. Izosimov et al., "Synthesis of Fault-Tolerant Schedules with Transparency/Performance Trade-offs for Distributed Embedded Systems", *Proc. Design Automation and Test in Europe Conference*, 706-711, 2006.

[12] 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.

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

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

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

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

[17] 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.

[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] C. R. Reevs, *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell Scientific Publications, 1993.

[20] J. Sosnowski, "Transient Fault Tolerance in Digital Systems", *IEEE Micro*, 14(1), 24-35, 1994.

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

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