DTU Library

# Synthesis of Fault-Tolerant Embedded Systems

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

*Published in:*
Design, Automation, and Test in Europe Conference

*Link to article, DOI:*
10.1109/DATE.2008.4484825

*Publication date:*
2008

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

# Synthesis of Fault-Tolerant Embedded Systems

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

[1]{petel|viaiz|zebpe}@ida.liu.se
Dept. of Computer and Information Science
Linköping University
SE–581 83 Linköping, Sweden

[2]Paul.Pop@imm.dtu.dk
Dept. of Informatics and Mathematical Modelling
Technical University of Denmark
DK-2800 Kongens Lyngby, Denmark

**Abstract**

*This work addresses the issue of design optimization for fault-tolerant hard real-time systems. In particular, our focus is on the handling of transient faults using both checkpointing with rollback recovery and active replication. Fault tolerant schedules are generated based on a conditional process graph representation. The formulated system synthesis approaches decide the assignment of fault-tolerance policies to processes, the optimal placement of checkpoints and the mapping of processes to processors, such that multiple transient faults are tolerated, transparency requirements are considered, and the timing constraints of the application are satisfied.*

## 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). Transient faults are the most common, and their number is continuously increasing due to high complexity, smaller transistor sizes, higher operational frequency, and lower voltage levels [5].

The rate of transient faults is often much higher compared to the one of permanent faults. Transient-to-permanent fault ratios can vary between 2:1 and 100:1 or higher [22].

From the fault tolerance point of view, transient faults and intermittent faults manifest themselves in a similar manner: they happen for a short time and then disappear without causing a permanent damage. Hence, fault tolerance techniques against transient faults are also applicable for tolerating intermittent faults and vice versa. Therefore, in this paper, we will refer to both types of faults as *transient faults* and we will talk about *fault tolerance against transient faults*, meaning tolerating both transient and intermittent faults.

Traditionally, hardware replication was used as a fault-tolerance technique against transient faults [21]. However, such solutions are very costly, in particular with increasing number of transient faults to be tolerated.

In order to reduce cost, other techniques are required such as software replication [3, 28], recovery with checkpointing [18, 27, 29], and re-execution [19]. However, if applied in a straightforward manner to an existing design, these techniques *introduce significant time overheads*, which can lead to unschedulable solutions. On the other hand, using faster components or a larger number of resources may not be affordable due to cost constraints. *Therefore, efficient design optimization techniques are required in order to meet time and cost constraints in the context of fault tolerant systems*.

Transient faults are also common for communication channels, even though, in this paper, we do not deal with them explicitly. Fault tolerance against multiple transient faults affecting communications has been studied and solutions such as a cyclic redundancy code (CRC) are implemented in communication protocols available on the market [10, 23].

Researchers have shown that schedulability of an application can be guaranteed for preemptive on-line scheduling under the presence of a single transient fault [1, 2, 12].

Liberato et al. [24] propose an approach for design optimization of monoprocessor systems in the presence of multiple transient faults and in the context of preemptive earliest-deadline-first (EDF) scheduling.

Hardware/software co-synthesis with fault tolerance is addressed in [6] where the minimum amount of additional hardware is determined in order to achieve a certain level of dependability. Xie et al. [28] propose a technique to decide how replicas can be selectively inserted into the application, based on process criticality. Introducing redundant processes into a pre-designed schedule is used in [4] in order to improve error detection. The above approaches only consider one single fault.

Kandasamy et al. [19] propose constructive mapping and scheduling algorithms for transparent re-execution on multiprocessor systems. The work was later extended with fault-tolerant transmission of messages on a time-division multiple access bus [20]. Both papers consider only one fault per computation node. Only process re-execution is used as a fault-tolerance policy.

Very few research work is devoted to global system optimization in the context of fault tolerance. For example, Pinello et al. [25] propose a heuristic for combining several static schedules in order to mask fault patterns. Multiple failures are addressed with active replication in [11] in order to guarantee a required level of fault tolerance and satisfy time constraints.

None of the previous work has considered fault-tolerance policies in the global context of system-level design for distributed embedded systems. Thus, we consider hard real-time safety-critical applications mapped on distributed embedded systems. Both the processes and the messages are scheduled using non-preemptive *quasi-static cyclic scheduling*. We consider two distinct fault-tolerance techniques: process-level *checkpointing* with rollback recovery [9], which provides time-redundancy, and active *replication* [26], which provides space-redundancy.

The main aspects of the work discussed here are:

- a quasi-static cyclic scheduling framework to schedule processes and messages, that can handle transparency/performance trade-offs imposed by the designer;
- mapping and fault tolerance policy assignment strategies for mapping of processes to computation nodes and assignment of a proper combination of fault tolerance techniques to processes, such that performance is maximized;
- an approach to the optimization of checkpoint distribution in rollback recovery.

## 2. System Architecture and Fault Model

We consider architectures composed of a set $\mathcal{N}$ of nodes which share a broadcast communication channel. Every *node* $N_i \in \mathcal{N}$

978-3-9810801-3-1/DATE08 © 2008 EDAA

consists, among others, of a communication controller and a CPU. The communications are scheduled statically based on schedule tables, and are fault-tolerant, using a TDMA based protocol, such as the Time Triggered Protocol (TTP) [23].

In this work we are interested in fault-tolerance techniques for transient faults. If permanent faults occur, we consider that they are handled using a technique such as hardware replication. Note that an architecture that tolerates $n$ permanent faults, will also tolerate $n$ transient faults. However, we are interested in tolerating a much larger number of transient faults than permanent ones, for which using hardware replication alone is too costly.

We have generalized the fault-model from [19] that assumes that only one single transient fault may occur on any node during an application execution. In our model, we consider that at most a given number $k$ of transient faults[1] may occur anywhere in the system during one operation cycle of the application. Thus, not only several transient faults may occur simultaneously on several processors, but also several faults may occur on the same processor.

## 3. Fault Tolerance Techniques

The error detection and fault-tolerance mechanisms are part of the software architecture. The software architecture, including the real-time kernel, error detection and fault-tolerance mechanisms are themselves fault-tolerant.

We use two mechanisms for tolerating faults: equidistant checkpointing with rollback recovery and active replication.

Once a fault is detected, a fault tolerance mechanism has to be invoked to handle this fault. The simplest fault tolerance technique to recover from fault occurrences is re-execution [19]. In re-execution, a process is executed again if affected by faults.

The time needed for the detection of faults is accounted for by the *error-detection overhead* $\alpha$. When a process is re-executed after a fault was detected, the system restores all initial inputs of that process. The process re-execution operation requires some time for this that is captured by the *recovery overhead* $\mu$.

### 3.1 Rollback Recovery with Checkpointing

The time overhead for re-execution can be reduced with more complex fault tolerance techniques such as *rollback recovery with checkpointing* [27, 29]. The basic principle of this technique is to restore the last non-faulty state of the failing process, i.e., to *recover* from faults. The last non-faulty state, or *checkpoint*, has to be saved in advance in the static memory and will be restored if the process fails. The part of the process between two checkpoints or between a checkpoint and the end of the process is called *execution segment*.

An example of rollback recovery with checkpointing is presented in Fig. 1. We consider process $P_1$ with the worst-case execution time of 60 ms and error-detection overhead $\alpha$ of 10 ms, as depicted in Fig. 1a. Fig. 1b presents the execution of $P_1$ in case no fault occurs, while Fig. 1c shows a scenario where a fault (depicted with a lightning bolt) affects $P_1$. In Fig. 1b, two checkpoints are inserted at equal intervals. The first checkpoint is the initial state of process $P_1$. The second checkpoint, placed in the middle of process execution, is for storing an intermediate process state. Thus, process $P_1$ is composed of two execution segments. We will name the $k$-th execution segment of process $P_i$ as $P_i^k$. Accordingly, the first execution segment of process $P_1$ is $P_1^1$ and its second segment is $P_1^2$. Saving process states, includ-

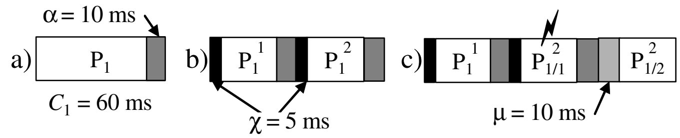



**Figure 1. Rollback Recovery with Checkpointing**

ing saving initial inputs, at checkpoints, takes an amount of time that is considered in the *checkpointing overhead* $\chi$, depicted as a black rectangle. In Fig. 1c, a fault affects the second execution segment $P_1^2$ of process $P_1$. This faulty segment is executed again starting from the second checkpoint. Note that the error-detection overhead $\alpha$ is not considered in the last recovery in the context of rollback recovery with checkpointing because, in this example, we assume that a maximum of one fault can happen.

We will denote the $j$-th execution of the $k$-th execution segment of process $P_i$ as $P_{i/j}^k$. Accordingly, the first execution of execution segment $P_1^2$ has the name $P_{1/1}^2$ and its second execution is named $P_{1/2}^2$. Note that we will not use the index $j$ if we only have one execution of a segment or a process, as, for example, $P_1$'s first execution segment $P_1^1$ in Fig. 1c.

When recovering, similar to re-execution, we consider a recovery overhead $\mu$, which includes the time needed to restore checkpoints. In Fig. 1c, the recovery overhead $\mu$, depicted with a light gray rectangle, is 10 ms for process $P_1$.

The fact that only a part of a process has to be restarted for tolerating faults, not the whole process, can considerably reduce the time overhead of rollback recovery with checkpointing compared to simple re-execution. Simple re-execution is a particular case of rollback recovery with checkpointing, in which a single checkpoint is applied, at process activation.

### 3.2 Active and Passive Replication

The disadvantage of recovery techniques is that they are unable to explore spare capacity of available computation nodes and, by this, to possibly reduce the schedule length. In contrast to rollback recovery and re-execution, *active and passive replication* techniques can utilize spare capacity of other computation nodes. Moreover, active replication provides the possibility of *spatial redundancy*, e.g. the ability to execute process replicas in parallel on different computation nodes.

In the case of active replication, all replicas are executed independently of fault occurrences. In the case of passive replication, also known as *primary-backup*, on the other hand, replicas are executed only if faults occur. In Fig. 2 we illustrate primary-backup and active replication. We consider process $P_1$ with the worst-case execution time of 60 ms and error-detection overhead $\alpha$ of 10 ms, see Fig. 2a. Process $P_1$ will be replicated on two computation nodes $N_1$ and $N_2$, which is enough to tolerate a single fault. We will name the $j$-th replica of process $P_i$ as $P_{i(j)}$.

In the case of active replication, illustrated in Fig. 2b, replicas

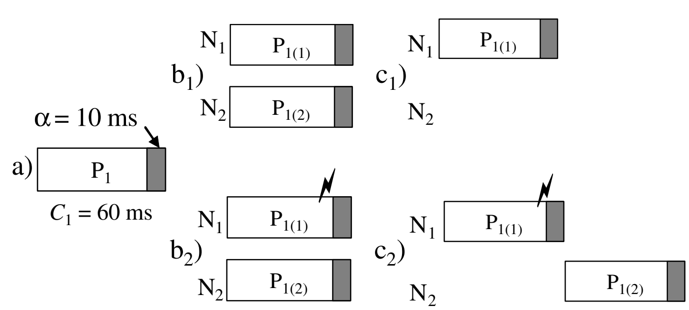



**Figure 2. Active Replication (b) and Primary-Backup (c)**

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

$P_{1(1)}$ and $P_{1(2)}$ are executed in parallel, which, in this case, improves system performance. However, active replication occupies more resources compared to primary-backup because $P_{1(1)}$ and $P_{1(2)}$ have to run even if there is no fault, as shown in Fig. 2b$_1$. In the case of primary-backup (Fig. 2c), the "backup" replica $P_{1(2)}$ is activated only if a fault occurs in $P_{1(1)}$. However, if faults occur, primary-backup takes more time to complete, compared to active replication, as shown in Fig. 2c$_2$ and Fig. 2b$_2$.

In our work, we are interested in active replication. This type of replication provides the possibility of spatial redundancy, which is lacking in rollback recovery. Moreover, rollback recovery with a single checkpoint is, in fact, a restricted case of primary-backup where replicas are only allowed to execute on the same computation node with the original process.

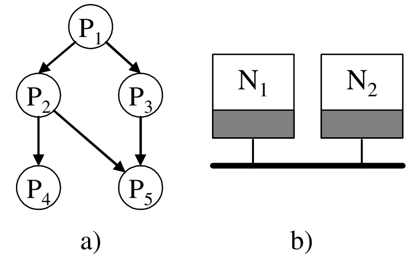
### 3.3 Transparency

Tolerating transient faults leads to many alternative execution scenarios, which are dynamically adjusted in the case of fault occurrences. The number of execution scenarios grows exponentially with the number of processes and the number of tolerated transient faults. In order to debug, test, or verify the system, all its execution scenarios have to be taken into account. Therefore, debugging, verification and testing become very difficult. A possible solution against this problem is *transparency.*

Originally, Kandasamy et al. [19] propose *transparent* re-execution, where recovering from a transient fault on one computation node is hidden from other nodes. In our work we apply a more flexible notion of transparency by allowing the designer to declare arbitrary processes and messages as frozen (see Section 4). Transparency has the advantage of fault containment and increased debugability. Since the occurrence of faults in certain process does not affect the execution of other processes, the total number of execution scenarios is reduced. Therefore, less number of execution alternatives have to be considered during debugging, testing, and verification. However, transparency can increase the worst-case delay of processes, reducing performance of the embedded system [14, 16].

## 4. Application Model

We consider a set of real-time periodic applications $\mathcal{A}_k$. Each application $\mathcal{A}_k$ is represented as an acyclic directed graph $\mathcal{G}_k(\mathcal{V}_k,\mathcal{E}_k)$. Each process graph is executed with the period $T_k$. The graphs are merged into a single graph with a period $T$ obtained as the least common multiple (LCM) of all application periods $T_k$. This graph corresponds to a virtual application $\mathcal{A}$, captured as a directed, acyclic graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$. Each node $P_i \in \mathcal{V}$ represents a process and each edge $e_{ij} \in \mathcal{E}$ from $P_i$ to $P_j$ indicates that the output of $P_i$ is the input of $P_j$.




WCET

| | $N_1$ | $N_2$ |
|---|---|---|
| $P_1$ | 20 | 30 |
| $P_2$ | 40 | 60 |
| $P_3$ | 60 | X |
| $P_4$ | 40 | 60 |
| $P_5$ | 40 | 60 |

c)

**Figure 3. A Simple Application and a Hardware Architecture**

Processes are non-preemptable. They send their output values encapsulated in messages, when completed. All required inputs have to arrive before activation of the process. Fig. 3a shows an application represented as a graph composed of five nodes.

Time constraints are imposed with a global hard deadline $D$, at which the application $\mathcal{A}$ *has to* complete. Some processes may also have local deadlines $d_{local}$.

The mapping of an application process is determined by a function $\mathcal{M}$: $\mathcal{V} \rightarrow \mathcal{N}$, where $\mathcal{N}$ is the set of nodes in the architecture. The mapping will be determined as part of the design optimization. For a process $P_i$, $\mathcal{M}(P_i)$ is the node to which $P_i$ is assigned for execution. Each process can potentially be mapped on several nodes. Let $\mathcal{N}_{P_i} \subseteq \mathcal{N}$ be the set of nodes to which $P_i$ can potentially be mapped. We consider that for each $N_k \in \mathcal{N}_{P_i}$, we know the worst-case execution time (WCET) $C_{P_i}^{N_k}$ of process $P_i$, when executed on $N_k$.

Fig. 3c shows the worst-case execution times of processes of the application depicted in Fig. 3a when executed on the architecture in Fig. 3b. For example, $P_2$ has the worst-case execution time of 40 ms if mapped on computation node $N_1$ and 60 ms if mapped on node $N_2$. By "X" we show mapping restrictions. For example, process $P_3$ cannot be mapped on computation node $N_2$.

In the case of processes mapped on the same node, message transmission time between them is accounted for in the worst-case execution time of the sending process. If processes are mapped on different nodes, then messages between them are sent through the communication network. We consider that the worst-case size of messages is given, which, implicitly, can be translated into a worst-case transmission time on the bus.

The combination of fault-tolerance policies to be applied to each process (Fig. 4) is given by four functions:

- $\mathcal{P}$: $\mathcal{V} \rightarrow$ {*Replication*,*Checkpointing*,*Replication&Checkpointing*} determines whether a process is replicated, checkpointed, or replicated and checkpointed.
- The function $\mathcal{Q}$: $\mathcal{V} \rightarrow$ N indicates the number of replicas for each process. For a certain process $P_i$, and considering $k$ the maximum number of faults, if $\mathcal{P}(P_i)$ = *Replication*, then $\mathcal{Q}(P_i)$ = $k$; if $\mathcal{P}(P_i)$ = *Checkpointing*, then $\mathcal{Q}(P_i)$ = 0; if $\mathcal{P}(P_i)$ =

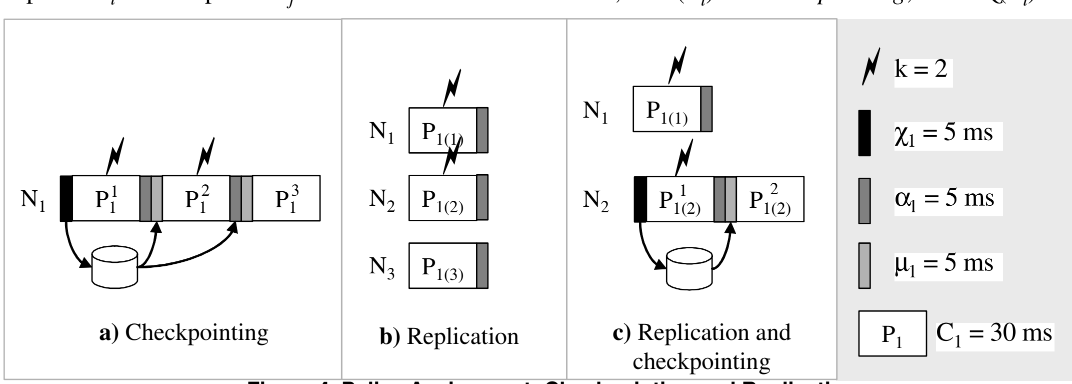



**Figure 4. Policy Assignment: Checkpointing and Replication**

*Replication & Checkpointing*, then $0 < \mathcal{Q}(P_i) < k$.

- Let $\mathcal{V}_R$ be the set of replica processes introduced into the application. Replicas can be also checkpointed, if necessary. Function $\mathcal{R}$: $\mathcal{V} \cup \mathcal{V}_R \rightarrow$N determines the number of possible recoveries for each process or replica. In Fig. 4a, $\mathcal{P}(P_1)$ = *Checkpointing*, $\mathcal{R}(P_1) = 2$. In Fig. 4b, $\mathcal{P}(P_1)$ = *Replication*, $\mathcal{R}(P_{1(1)}) = \mathcal{R}(P_{1(2)}) = \mathcal{R}(P_{1(3)}) = 0$. In Fig. 4c, $\mathcal{P}(P_1)$ = *Replication & Checkpointing*, $\mathcal{R}(P_{1(1)}) = 0$ and $\mathcal{R}(P_{1(2)}) = 1$.
- Function $\mathcal{X}$: $\mathcal{V} \cup \mathcal{V}_R \rightarrow$N indicates the number of checkpoints to be applied to processes in the application and the replicas in $\mathcal{V}_R$. We consider equidistant checkpointing, thus the checkpoints are equally distributed throughout the execution time of the process. If process $P_i \in \mathcal{V}$ or replica $P_{i(j)} \in \mathcal{V}_R$ is not checkpointed, then we have $\mathcal{X}(P_i) = 0$ or $\mathcal{X}(P_{i(j)}) = 0$, respectively.

Each process $P_i \in \mathcal{V}$, besides its worst-case execution time $C_i$, is characterized by an error detection overhead $\alpha_i$, a recovery overhead $\mu_i$, and checkpointing overhead $\chi_i$.

The transparency requirements imposed by the user are captured by a function $\mathcal{T}$: $\mathcal{V} \rightarrow$ {*frozen*, *not_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. If $\mathcal{T}(v_i)$ = *frozen*, our scheduling algorithm will handle this transparency requirements by allocating the same start time for $v_i$ in all the alternative fault-tolerant schedules of application $\mathcal{A}$.

## 5. Fault Tolerant Schedules

Our approach to the generation of fault-tolerant system schedules is based on the fault-tolerant conditional process graph (FT-CPG) representation, an application of Conditional Process Graphs [7, 8]. The final schedules are produced as a set of schedule tables that are capturing the alternative execution scenarios corresponding to possible fault occurrences.

### 5.1 Fault Tolerant Conditional Process Graph

A FT-CPG captures alternative execution scenarios in the case of possible fault occurrences. A fault occurrence is captured as a condition, which is *true* if the fault happens and *false* otherwise.

A FT-CPG is a directed acyclic graph $G(V_P \cup V_C \cup V_T, E_S \cup E_C)$. We denote a node in the FT-CPG with $P_i^m$ that will correspond to the $m^{th}$ copy of process $P_i \in \mathcal{A}$. A node $P_i^m \in V_P$ with simple edges at the output is a regular node. A node $P_i^m \in V_C$ with *conditional edges* at the output is a *conditional process* that produces a condition. A node $v_i \in V_T$ is a *synchronization node* and represents the synchronization point corresponding to a frozen process or message (i.e., $\mathcal{T}(v_i)$ = *frozen*). We denote with $P_i^S$ the synchronization node of process $P_i \in \mathcal{A}$ and with $m_i^S$ the synchronization node of message $m_i \in \mathcal{A}$. Synchronization nodes take zero time to execute.

$E_S$ and $E_C$ are the sets of simple and conditional edges, respectively. An edge $e_{ij}^{mn} \in E_S$ from $P_i^m$ to $P_j^n$ indicates that the output of $P_i^m$ is the input of $P_j^n$. Synchronization nodes $P_i^S$ and $m_i^S$ are also connected through edges to regular and conditional processes and other synchronization nodes.

Edges $e_{ij}^{mn} \in E_C$ are *conditional edges* and have an associated condition value. The condition value produced is "true" (denoted with $F_{P_i^m}$) if $P_i^m$ experiences a fault, and "false" (denoted with $\overline{F}_{P_i^m}$) if $P_i^m$ does not experience a fault. Alternative paths starting from such a process, which correspond to complementary values of the condition, are disjoint[1]. Regular and conditional processes are activated when all their inputs have arrived. A

1. They can only meet in a synchronization node.

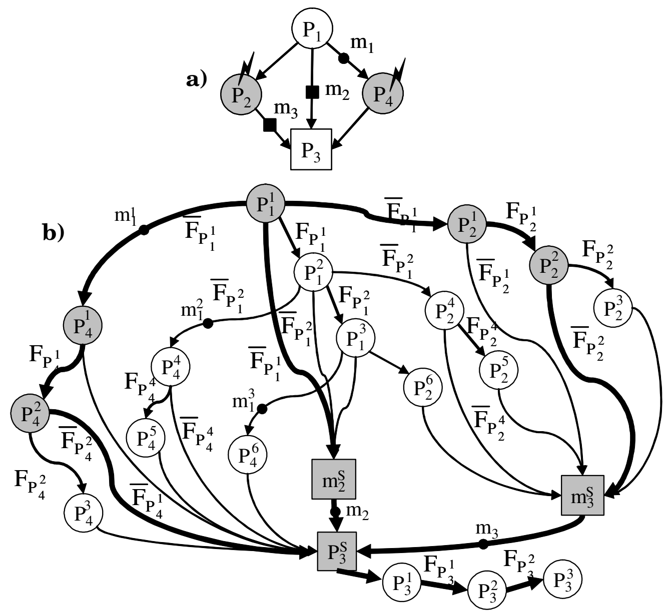



**Figure 5. Fault Tolerant Conditional Process graph**

synchronization node can be activated after inputs coming on one of the alternative paths, corresponding to a particular fault scenario, have arrived.

Fig. 5a depicts an application $\mathcal{A}$ modelled as a process graph $\mathcal{G}$, which can experience at most two faults (for example, as in the figure, during the execution of processes $P_2$ and $P_4$). Transparency requirements are depicted with rectangles on the application graph, where process $P_3$, message $m_2$ and message $m_3$ are set to be frozen. For scheduling purposes we will convert the application $\mathcal{A}$ to a fault-tolerant conditional process graph (FT-CPG) $G$, represented in Fig. 5b. In an FT-CPG the fault occurrence information is represented as *conditional edges* and the frozen processes/messages are captured using *synchronization nodes*. One of the conditional edges is $P_1^1$ to $P_4^1$ in Fig. 5b, with the associated condition $\overline{F}_{P_1^1}$ denoting that $P_1^1$ has no faults. Message transmission on conditional edges takes place only if the associated condition is satisfied.

The FT-CPG in Fig. 5b captures all the fault scenarios that can happen during the execution of application $\mathcal{A}$ in Fig. 5a. The subgraph marked with thicker edges and shaded nodes in Fig. 5b captures the execution scenario when processes $P_2$ and $P_4$ experience one fault each. 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 $\overline{F}_{P_4^1}$ (no-fault). The transparency requirement that, for example, $P_3$ has to be frozen, is captured by the synchronization node $P_3^S$ inserted before the conditional edge with copies of process $P_3$. In Fig. 5b, process $P_1^1$ is a conditional process because it "produces" condition $F_{P_1^1}$, while $P_1^3$ is a regular process.

### 5.2 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 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. 6 presents the schedules for the nodes

| $N_1$ | $true$ | $F_{P_1^1}$ | $\overline{F}_{P_1^1}$ | $F_{P_1^1} \wedge F_{P_1^2}$ | $F_{P_1^1} \wedge \overline{F}_{P_1^2}$ | $F_{P_1^1} \wedge \overline{F}_{P_1^2} \wedge F_{P_2^4}$ | $F_{P_1^1} \wedge \overline{F}_{P_1^2} \wedge \overline{F}_{P_2^4}$ | $\overline{F}_{P_1^1} \wedge F_{P_2^1}$ | $\overline{F}_{P_1^1} \wedge F_{P_2^1} \wedge F_{P_2^2}$ | $\overline{F}_{P_1^1} \wedge F_{P_2^1} \wedge \overline{F}_{P_2^2}$ | $\overline{F}_{P_1^1} \wedge \overline{F}_{P_2^1}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $P_1$ | 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 | | | | | | | | | |

| $N_2$ | $true$ | $F_{P_1^1}$ | $\overline{F}_{P_1^1}$ | $F_{P_1^1} \wedge F_{P_1^2}$ | $F_{P_1^1} \wedge \overline{F}_{P_1^2}$ | $F_{P_1^1} \wedge \overline{F}_{P_1^2} \wedge F_{P_4^4}$ | $F_{P_1^1} \wedge \overline{F}_{P_1^2} \wedge \overline{F}_{P_4^4}$ | $\overline{F}_{P_1^1} \wedge F_{P_4^1}$ | $\overline{F}_{P_1^1} \wedge F_{P_4^1} \wedge F_{P_4^2}$ | $\overline{F}_{P_1^1} \wedge F_{P_4^1} \wedge \overline{F}_{P_4^2}$ | $\overline{F}_{P_1^1} \wedge \overline{F}_{P_4^1}$ | $F_{P_3^1}$ | $F_{P_3^1} \wedge F_{P_3^2}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $P_3$ | | | | 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 6. Schedule Tables**

$N_1$ and $N_2$ produced for the FT-CPG in Fig. 5. 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 computation nodes. 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 $N_1$, 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 $\overline{F}_{P_1^1}$ 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 computation node to all other computation nodes. This broadcast is scheduled as soon as possible on the communication channel, and is considered together with the scheduling of the messages.

In [13, 14, 17] we have presented several algorithms for the synthesis of fault tolerant schedules. They are allowing for various trade-offs between the worst case schedule length, the size of the schedule tables, the degree of transparency, and the duration of the schedule generation procedure.

## 6. Fault Tolerant System Design

By policy assignment we denote the decision whether a certain process should be checkpointed or replicated, or a combination of the two should be used. Mapping a process means placing it on a particular node in the architecture.

There are cases when the policy assignment decision is taken based on the experience of the designer, considering aspects like the functionality implemented by the process, the required level of reliability, hardness of the constraints, legacy constraints, etc. Many processes, however, do not exhibit particular features or requirements which obviously lead to checkpointing or replication. Decisions concerning the policy assignment for these processes can lead to various trade-offs concerning, for example, the schedulability properties of the system, the amount of communication exchanged, the size of the schedule tables, etc.

For part of the processes in the application, the designer might have already decided their mapping. For example, certain processes, due to constraints like having to be close to sensors/actuators, have to be physically located in a particular hardware unit. For the rest of the processes (including the replicas) their mapping is decided during design optimization.

Thus, our problem formulation for mapping and policy assignment with checkpointing is as follows:

- As an input we have an application $\mathcal{A}$ (Section 4) and a system consisting of a set of nodes $\mathcal{N}$ connected to a bus $B$ (Section 2).
- The parameter $k$ denotes the maximum number of transient faults that can appear in the system during one cycle of execution.

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

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

1. finding a fault tolerance policy assignment, given by $\mathcal{F} = <\mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathcal{X}>$, for each process $P_i$ (see Section 4) in the application $\mathcal{A}$, for which the fault-tolerance policy has not been a priory set by the designer; this also includes the decision on the number of checkpoints $\mathcal{X}$ for each process $P_i$ in the application $\mathcal{A}$ and each replica in $\mathcal{V}_R$;
2. deciding on a mapping $\mathcal{M}$ for each unmapped process $P_i$ in the application $\mathcal{A}$;
3. deciding on a mapping $\mathcal{M}$ for each unmapped replica in $\mathcal{V}_R$;
4. deriving the set $\mathcal{S}$ of schedule tables.

Based on the scheduling approaches described in [13, 14, 17] we have developed several heuristics that are solving the above formulated design problem. In particular, we have addressed the problem of fault-tolerant application mapping in [16], and the issue of checkpointing optimization in [15]. An approach to optimal fault tolerance policy assignment has been presented in [13].

The graph in Fig. 7 illustrates the efficiency of the mapping and

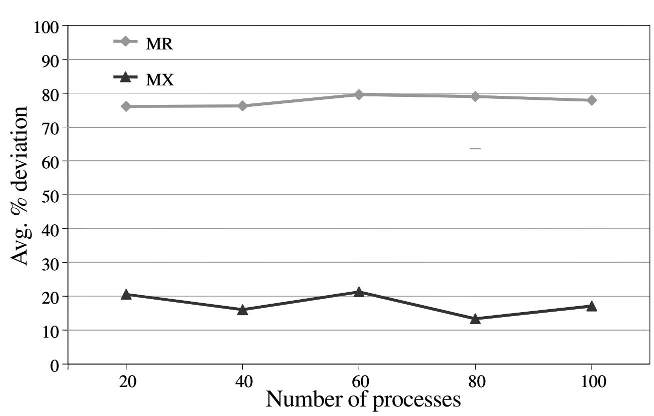



**Figure 7. Efficiency of fault tolerance policy assignment**

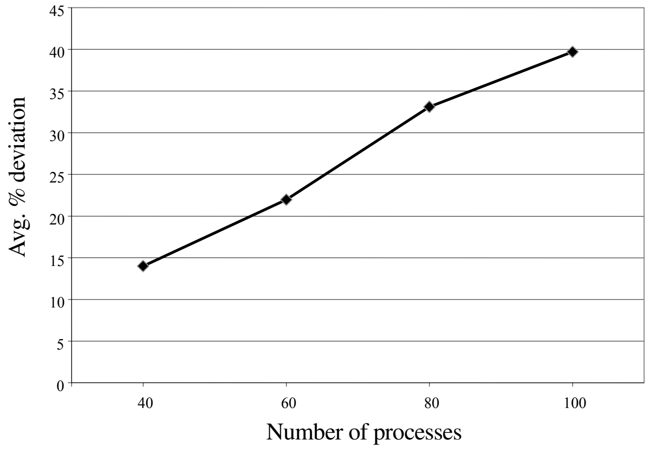



**Figure 8. Efficiency of checkpointing optimization**

fault tolerance policy assignment approach described in [13]. Experiments with applications consisting of 20 to 100 processes implemented on architectures consisting of 2 to 6 nodes have been performed. The number of tolerated faults was between 3 and 7. The parameter we are interested in is the fault tolerance overhead (FTO) which represents the percentage increase of the schedule length due to fault tolerance considerations. We obtained the FTO by comparing the schedule length obtained using our techniques with the length of the schedules using the same (mapping and scheduling) techniques but with ignoring the fault tolerance issues. As a baseline in Fig. 7 we use the FTO produced by our approach proposed in [13], which optimizes the process mapping and also assigns a fault tolerance policy (re-execution or replication) to tasks such as the schedule length is minimized. We compared our approach with two extreme approaches: MX that only considers reexecution and MR which only relies on replication for tolerating faults. As the graph shows, optimizing the assignment of fault tolerance policies leads to results that are, on average, 77% and 17,6% better than MR and MX, respectively.

In Fig. 8 we illustrate the efficiency of our checkpointing optimization technique proposed in [15]. This technique extends the one proposed in [13] by considering re-execution with checkpointing and by proposing an approach to optimization of the number of checkpoints. The baseline for the graph in Fig. 8 is the FTO produced by optimizing the number of checkpoints using a technique proposed in [27]. This technique determines the optimal number of checkpoints considering each process in isolation, as a function of the checkpointing overhead (which depends on the time needed to create a checkpoint). However, calculating the number of checkpoints for each individual process will not produce a solution which is globally optimal for the whole application. In Fig. 8 we show the average percentage deviation of the FTO obtained with the system optimization technique proposed in [15] from the baseline obtained with the checkpoint optimization proposed in [27] (in this graph, larger deviation means smaller overhead).

## 7. Conclusions

In this paper we have addressed the issue of design optimization for real-time systems with fault tolerance requirements. In particular, we have emphasized the problem of transient faults since their number is continuously increasing with new electronic technologies. We have shown that efficient system-level design optimization techniques are required in order to meet the imposed design constraints for fault tolerant embedded systems in the context of a limited amount of available resources.

## References


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

[2] A. Burns, R.I. Davis, S. Punnekkat, "Feasibility Analysis for Fault-Tolerant Real-Time Task Sets", *Euromicro Worksh. on Real-Time Systems*, 29-33, 1996.

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

[4] J. Conner, Y. Xie, M. Kandemir, G. Link, R. Dick, "FD-HGAC: A Hybrid Heuristic/Genetic Algorithm Hardware/Software Co-synthesis Framework with Fault Detection", *ASP-DAC Conf.*, 709-712, 2005.

[5] C. Constantinescu, "Trends and Challanges in VLSI Circuit Reliability", *IEEE Micro*, 23(4), 14-19, 2003.

[6] B. P. Dave, N. K. Jha, "COFTA: Hardware-Software Co-Synthesis of Heterogeneous Distributed Embedded System for Low Overhead Fault Tolerance", *IEEE Trans. on Computers*, 48(4), 417-441, 1999.

[7] P. Eles, K. Kuchcinski, Z. Peng, P. Pop, A. Doboli, "Scheduling of Conditional Process Graphs for the Synthesis of Embedded Systems", *DATE Conf*, 132-138, 1998.

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

[9] E. N. Elnozahy, L. Alvisi, Y. M. Wang, D. B. Johnson, "A Survay of Rollback-Recovery Protocols in Message-Passing Systems", *ACM Computing Surveys*, 34(3), 375-408, 2002.

[10] FlexRay Consortium, "FlexRay Protocol Specification", Ver. 2.0, 2004.

[11] A. Girault, H. Kalla, M. Sighireanu, Y. Sorel, "An Algorithm for Automatically Obtaining Distributed and Fault-Tolerant Static Schedules", *Int. Conf. on Dependable Syst. and Netw.*, 159-168, 2003.

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

[13] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Design Optimization of Time- and Cost-Constrained Fault-Tolerant Distributed Embedded Systems", *DATE Conf.*, 864-869, 2005.

[14] V. Izosimov, P. Pop, P. Eles, Z. Peng, "Synthesis of Fault-Tolerant Schedules with Transparency/Performance Trade-offs for Distributed Embedded Systems", *DATE Conf.*, 706-711, 2006.

[15] V. Izosimov, P. Pop, P. Eles, Z. Peng, "Synthesis of Fault-Tolerant Embedded Systems with Checkpointing and Replication", *IEEE Intl. Worksh. on Electron. Design, Test & Appl. (DELTA),* 440-447, 2006.

[16] V. Izosimov, P. Pop, P. Eles, Z. Peng, "Mapping of Fault-Tolerant Applications with Transparency on Distributed Embedded Systems", *Euromicro Conf. on Digital System Design (DSD),* 313-320, 2006.

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

[18] Jie Xu, B. Randell, "Roll-Forward Error Recovery in Embedded Real-Time Systems", *Int. Conf. on Parallel and Distr. Syst.*, 414-421, 1996.

[19] 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.

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

[21] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, J. Reisinger, "Tolerating Transient Faults in MARS", *20th Int. Symp. on Fault-Tolerant Computing*, 466-473, 1990.

[22] H. Kopetz, R. Obermaisser, P. Peti, N. Suri, "From a Federated to an Integrated Architecture for Dependable Embedded Real-Time Systems", *Tech. Rep. 22, TU Vienna*, 2003.

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

[24] F. Liberato, R. Melhem, and D. Mosse, "Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard Real-Time Systems", *IEEE Trans. on Computers*, 49(9), 906-914, 2000.

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

[26] S. Poldena, "Fault Tolerant Systems - The Problem of Replica Dterminism", *Kluwer*, 1996.

[27] S. Punnekkat, A. Burns, R. Davis, "Analysis of Checkpointing for Real-Time Systems", *Real-Time Systems Journal*, 20(1), 83-102, 2001.

[28] Y. Xie, L. Li, M. Kandemir, N. Vijaykrishnan, and M.J. Irwin, "Reliability-Aware Co-synthesis for Embedded Systems", *Proc. 15th IEEE Intl. Conf. on Appl.-Spec. Syst., Arch. and Proc.*, 41-50, 2004.

[29] Ying Zhang and K. Chakrabarty, "A Unified Approach for Fault Tolerance and Dynamic Power Management in Fixed-Priority Real-Time Embedded Systems", *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 25(1), 111-125, 2006.