Scheduling and Optimization of Fault-Tolerant Embedded Systems with Transparency/ Performance Trade-Offs

VIACHESLAV IZOSIMOV
Embedded Intelligent Solutions (EIS) By Semcon AB
PAUL POP
Technical University of Denmark
and
PETRU ELES AND ZEBO PENG
Linköping University

In this paper, we propose a strategy for the synthesis of fault-tolerant schedules and for the mapping of fault-tolerant applications. Our techniques handle transparency/performance trade-offs and use the fault-occurrence information to reduce the overhead due to fault tolerance. Processes and messages are statically scheduled, and we use process re-execution for recovering from multiple transient faults. We propose a fine-grained transparent recovery, where the property of transparency can be selectively applied to processes and messages. Transparency hides the recovery actions in a selected part of the application so that they do not affect the schedule of other processes and messages. While leading to longer schedules, transparent recovery has the advantage of both improved debuggability and less memory needed to store the fault-tolerant schedules.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems - *Real-time and Embedded Systems*; C.4 [**Computer Systems Organization**]: Performance of Systems - *Design Studies; Fault Tolerance*; D.2.5 [**Software**]: Testing and Debugging; D.4.5 [**Software**]: Reliability - *Checkpoint/Restart*; C.1.4 [**Computer Systems Organization**]: Parallel Architectures - *Distributed Architectures*; D.4.1 [**Software**]: Process Management - *Scheduling*; G.1.6 [**Mathematics of Computing**]: Optimization - *Global Optimization*
General Terms: Design, Performance, Reliability
Additional Key Words and Phrases: Fault-tolerant embedded systems, safety-critical applications, design optimization, transient faults, intermittent faults, debuggability, real-time scheduling, conditional scheduling, process mapping

---

1. INTRODUCTION

Modern embedded systems are complex computer systems with sophisticated software running on often distributed hardware platforms. Such systems can provide very high

This research was partially supported by the Swedish National Graduate School in Computer Science (CUGS).
This work was done while Viacheslav Izosimov was working at Dept. of Computer and Information Science at Linköping University in Sweden.
Authors' addresses: Viacheslav Izosimov, Embedded Intelligent Solutions (EIS) By Semcon AB, Box 407, SE-58104 Linköping, Sweden; Paul Pop, Dept. of Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Kongens Lyngby, Denmark; Petru Eles and Zebo Peng, Dept. of Computer and Information Science, Linköping University, SE-58183 Linköping, Sweden.

Received April 2009; revised February 2010; accepted October 2010.

levels of service and are replacing mechanical and hydraulic parts of control systems in aircraft, automobiles, production lines, switching systems, and medical equipment. They can be responsible for safety-critical operations in, for example, automatic flight control, electronic steering, or car braking systems.

Safety-critical embedded systems have to deliver correct results even in the presence of faults. Faults can be permanent (e.g., damaged links or microcontrollers), transient, and intermittent [Koren and Krishna, 2007]. Transient and intermittent faults[1] (or "soft errors") appear for a short time, cause miscalculation in logic [Rossi et al., 2005] and/or corruption of data, and then disappear without physical damage to the circuit. Causes of transient faults can be radiation [Velazco et al., 2007; May and Woods, 1978], electromagnetic interference [Strauss et al., 2006; Wang, 2003], lightning storms [Heine et al., 2005], crosstalk [Metra et al., 1998], temperature variations [Wei et al., 2004], and power supply fluctuations [Junior et al., 2004]. In this paper we will deal with transient faults. Transient faults have become one of the main concerns in the design of modern embedded systems due to the increased levels of integration in semiconductors, e.g. smaller transistor sizes, higher frequencies, and lower voltages [Constantinescu, 2003; Maheshwari et al., 2004; Hareland et al., 2001; Shivakumar et al., 2002]. According to recent studies [Kopetz et al., 2004; Shivakumar et al., 2002], the rate of transient-to-permanent faults can be 100:1 or even higher.

Traditionally, transient faults have been addressed with hardware replication [Kopetz et al., 1990; Alstrom and Torin, 2001; Poledna, 1995; Claesson et al., 1998]. However, solutions based on the replication of hardware are very costly, in particular, with the increasing number of transient faults that have to be tolerated. In order to reduce the cost, other techniques are required, such as re-execution [Kandasamy et al., 2003a], replication [Xie et al., 2004; Xie et al., 2007; Chevochot and Puaut, 1999] and recovery with checkpointing [Punnekkat and Burns, 1997; Zhang and Chakrabarty, 2006; Orailoglu and Karri, 1994; Xu and Randell, 1996; Krishna and Singh, 1993; Lee et al., 1999; Melhem et al., 2004; Ayav et al., 2008]. The above techniques can also tolerate some of the software errors that manifest themselves as transient faults, for example, Heisenbugs [Kopetz et al., 2004], caused by wrongly initialized data or synchronization-related problems.

Safety-critical embedded systems have to satisfy cost and performance constraints besides reliability requirements. For example, automotive applications that are responsible for such safety-critical functions as braking or stabilization have to be fault-

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

tolerant and, at the same time, meet cost and timing constraints. However, re-execution, replication and recovery with checkpointing, if applied in a straightforward manner, will lead to significant time overheads and, hence, to solutions that do not meet performance constraints. Thus, faster components or more resources will be demanded to satisfy performance constraints, which, on the other hand, may not be acceptable due to cost limitations. Therefore, efficient design approaches are needed to satisfy cost and timing requirements imposed on fault-tolerant embedded systems. Researchers have proposed design strategies for the synthesis of fault-tolerant embedded systems in the past years. Liberato et al. [2000] have proposed an approach for design optimization of monoprocessor systems in the presence of multiple transient faults and in the context of pre-emptive earliest-deadline-first (EDF) scheduling. Hardware/software co-synthesis with fault tolerance has been addressed in [Srinivasan and Jha, 1995] in the context of event-driven scheduling. Xie et al. [2004] have proposed a technique to decide how replicas are selectively inserted into the application, based on process criticality. Ayav et al. [2008] have achieved fault tolerance for real-time programs with automatic transformations, where recovery with checkpointing is used to tolerate one single fault at a time. Power-related optimization issues of fault-tolerant embedded systems have been studied in [Zhang and Chakrabarty, 2006; Melhem et al., 2004]. Ying Zhang et al. [2006] have studied fault tolerance and dynamic power management in the context of message-passing distributed systems. Fault tolerance has been applied on top of a pre-designed system, whose process mapping and scheduling ignore the fault tolerance issue. Melhem et al. [2004] have considered checkpointing for rollback recovery in the context of online earliest-deadline-first (EDF) scheduling on a monoprocessor embedded system. We have proposed a number of design optimization and scheduling techniques [Pop et al., 2009; Izosimov et al., 2005, 2006a, 2006b], including mapping and policy assignment, that are able to deliver efficient fault-tolerant embedded systems under limited amount of resources.

Fault tolerance techniques not only reduce the performance and increase the cost but also increase the complexity of embedded software. Compexity often leads to serious difficulties during debugging and testing of fault-tolerant embedded systems.

A common systematic approach for debugging embedded software is to insert observation points into software and hardware [Vranken et al., 1997; Tripakis, 2005; Savor and Seviora, 1997] for observing the system behavior under various circumstances. The observation points are usually inserted by an expert, or can be automatically injected based on statistical methods [Bourret et al., 2004]. In order to efficiently trace design

errors, the results produced with the observation points have to be easily monitored, even in the recovery scenarios against transient faults. Unfortunately, the number of recovery scenarios is often very high and, thus, monitoring observation points for all these scenarios is often infeasible. Moreover, due to the increased number of fault scenarios, the number of possible system states substantially increases. It results in a very complex system behavior that is difficult to test and verify. The overall number of possible recovery scenarios can be considerably reduced by restricting the system behavior, in particular, by introducing *transparency requirements*.

A transparent recovery scheme has been proposed in [Kandasamy et al., 2003a], where recovering from a transient fault on one computation node does not affect the schedule of any other node. In general, transparent recovery has the advantage of increased debuggability, where the occurrence of faults in a certain process does not affect the execution of other processes. This reduces the total number of execution scenarios. At the same time, with increased transparency, the amount of memory needed to store the schedules decreases. However, transparent recovery increases the worst-case delay of processes, potentially reducing the overall performance of the embedded system. Thus, efficient design optimization techniques are even more important in order to meet time and cost constraints in the context of fault-tolerant embedded systems with transparency requirements. However, to our knowledge, most of the design strategies proposed so far [Zhang and Chakrabarty, 2006; Xie et al., 2004; Pinello et al., 2008; Srinivasan and Jha, 1995; Melhem et al., 2004; Ayav et al., 2008]have not explicitly addressed the transparency requirements for fault tolerance. If at all addressed, these requirements have been applied, at a very coarse-grained level, to a whole computation node, as in the case of the original transparent re-execution proposed in [Kandasamy et al., 2003a]. In such a schema, system behavior can be observed only by monitoring messages sent to and out of the computation node. In this case, designers neither can observe intraprocessor process inputs nor observe intraprocessor messages. Moreover, the coarse-grained transparency also leads to unnecessary end-to-end delays since, by far, not all of the fixed interprocessor messages have to be observed.

In this paper we propose a design optimization strategy that efficiently handles more elaborate transparency requirements and, at the same time, provides schedulable fault-tolerant solutions under limited amount of resources.

1.1 Related Work

In the context of fault-tolerant real-time systems, researchers have tried to integrate fault tolerance techniques and task scheduling [Han et al., 2003; Bertossi and Mancini, 1994; Burns et al., 1996; Zhang and Chakrabarty, 2006; Xie et al., 2004; Wei et al., 2006]. Girault et al. [2003] have proposed a generic approach to address multiple failures with active replication. Ahn et al. [1997] have proposed a scheduling algorithm that generates efficient schedules with encapsulated primary-backup replicas against processor failures in a multiprocessor system. Passive replication has been used in [Al-Omari et al., 2001] to handle a single failure in multiprocessor systems so that timing constraints are satisfied. Liberato et al. [2000] have proposed an approach for design optimization of monoprocessor systems under presence of multiple transient faults. Conner et al. [2005] have introduced redundant processes into a pre-designed schedule to improve error detection. Hardware/software co-synthesis of fault-tolerant embedded systems has been addressed in [Srinivasan and Jha, 1995]. Ayav et al. [2008] have achieved fault tolerance for real-time programs with automatic transformations, where recovery with checkpointing is used to tolerate one single fault at a time. Xie et al. [2004; 2007] have proposed an approach to selectively insert replicas into the application with minimization of overall system criticality. Shye et al. [2007] have developed a process-level redundancy approach against multiple transient faults with active replication on multi-core processors. Power-related optimization issues of fault-tolerant embedded systems have been studied in [Zhang and Chakrabarty, 2006; Han and Li, 2005; Zhu et al., 2005; Melhem et al., 2004; Wei et al., 2006; Pop et al., 2007].

Kandasamy et al. [2003a] have proposed *transparent* re-execution, where recovering from a transient fault on one computation node is hidden (masked) from other nodes, i.e., they have considered *node-level transparency*. Later this work has been extended with fault-tolerant transmission of messages on the bus [Kandasamy et al., 2003b]. Pinello et al. [2004; 2008] have addressed primarily permanent faults and have proposed mapping and scheduling algorithms for embedded control software. In [Izosimov et al., 2005; Pop et al., 2009] we have extended the approach of [Kandasamy et al., 2003a] with active replication and checkpointing optimization, and have proposed a fault-tolerance policy assignment strategy to decide which fault tolerance technique, e.g., checkpointing, active replication, or their combination, is the best suited for a particular process in the application.

However, the scheduling approach in [Izosimov et al., 2005; Pop et al., 2009] is very limited in its capacity to accommodate various fault scenarios and, thus, will lead to

unnecessary long schedules. The approach also considers only coarse-grained, node-level transparency and cannot handle more elaborate transparency requirements applied to a particular process or message, or to a set of processes and messages. Such fine-grained transparency approaches are needed for, e.g., a selective insertion of observation points, where only a particular subset of processes and messages needs to be monitored [Bourret et al., 2004; Tripakis, 2005].

1.2 Contributions

In this paper, we present a novel algorithm for the synthesis of fault tolerant schedules that handles the transparency/performance trade-offs. The proposed algorithm not only handles fine-grained transparency, but, as the experimental results will show, also significantly reduces the schedule length compared to the previous scheduling approach [Izosimov et al., 2005; Pop et al., 2009].

A fine-grained approach to transparency, proposed in this paper, handles transparency requirements at the application level instead of resource level, selectively applying transparency to a particular process or message, or to a set of processes and messages. Thus, our fine-grained approach to transparency offers the designer the opportunity to gradually trade-off between debuggability and memory requirements on one side, and performance on the other side.

Our approach makes use of the fault-occurrence information in order to adapt schedules to the current fault scenario and, thus, reduce the overhead due to fault tolerance. We use a fault-tolerant process graph representation (FTPG) 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 an FTPG scheduling problem.

In this work, we also present an optimization algorithm that produces a mapping of processes on computation nodes such that the application is schedulable and the fault tolerance and transparency properties imposed by the designer are satisfied.

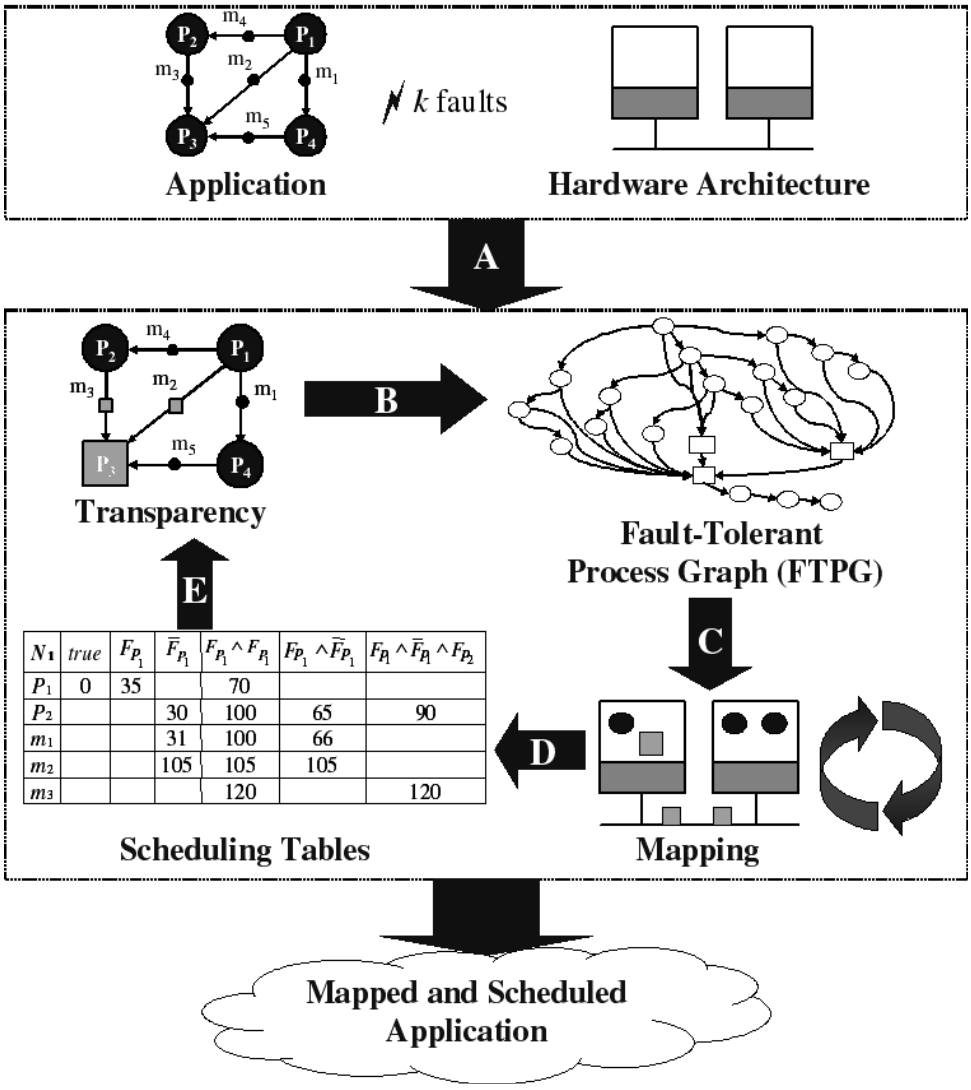
2. OVERALL SYNTHESIS FLOW

Our overall synthesis flow is outlined in Fig. 1. The application is modelled as a set of processes communicating using messages, which runs on the hardware architecture composed of a set of computation nodes connected to a communication bus (as described in Section 3). The fault tolerance and real-time constraints, such as the maximum number k of transient faults and process deadlines, are provided as input.

The actual synthesis and optimization is performed in several steps:

(A) The designer introduces transparency requirements, by selecting a set of processes and messages to be frozen. For example, designers can select important data communications as frozen for observing process inputs and outputs and for evaluation of timing properties of processes and messages.

(B) The application, with introduced transparency requirements, is translated into a fault-tolerant process graph (FTPG). The FTPG representation, presented in Section 7.1, captures the transparency properties and all possible combinations of fault occurrences.



N_1	*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

Fig. 1. Overall Synthesis Flow.

(C) The FTPG is passed over to the mapping optimization algorithm (Section 6.3). The algorithm optimizes the placement of application processes on the computation nodes and uses as a cost function the estimated schedule length (Section 7.4).

(D) Considering the mapping solution produced in the previous step, a fault-tolerant conditional schedule is synthesized as a set of schedule tables (Section 7.3). A distributed run time scheduler will use these schedule tables for execution of the application processes on the computation nodes.

(E) If the application is unschedulable, the designer has to change the transparency setup, re-considering the transparency/performance trade-offs.

The rest of the paper is organized as follows. Section 3 presents our application and system model. Section 4 introduces our fault model for multiple transient faults. Section 5 introduces transparency and illustrates the performance/transparency trade-offs on a set of motivational examples. Section 6 presents our problem formulation and overall design optimization strategy. Section 7 introduces the FTPG representation and presents our conditional scheduling algorithm for synthesis of fault-tolerant schedules as well as our schedule length estimation heuristic. The proposed scheduling and mapping algorithms are evaluated on a set of synthetic applications and a real-life example in Section 8. Conclusions are presented in Section 9.

3. APPLICATION AND SYSTEM MODEL

We consider a set of real-time applications. Each application A_k is represented as an acyclic directed graph $G_k(V_k, E_k)$. Each process graph G_k is executed with period T_k. The graphs for all applications are merged into a single graph with a period T obtained as a least common multiple (LCM) of all periods T_k [Pop et al., 2004]. This graph corresponds to a virtual application A, represented as a directed acyclic graph $G(V, E)$. Each node $P_i \in V$ represents one process. An edge $e_{ij} \in E$ from P_i to P_j indicates that the output of P_i is the input of P_j.

Processes are not preempted during their execution. A process can be activated after all its inputs have arrived. The process issues its outputs, encapsulated in messages, when it completes.

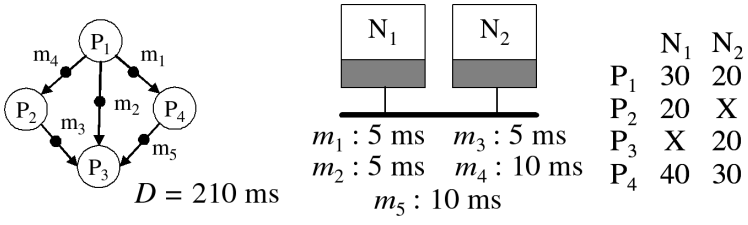
Time constraints are imposed with a global hard deadline $D \leq T$, at which the application A *has* to complete. In addition, processes can be associated with individual deadlines. An individual hard deadline d_i of a process P_i is modelled as a dummy node

inserted into the application graph with the execution time $C_{dummy} = D - d_i$ [Pop et al., 2004]. This dummy node however, is not allocated to any resource.

We consider that the application is running on a set of computation nodes N connected to a bus B. The mapping of processes in the application is determined by a function $M: V \rightarrow N$. For a process $P_i \in V$, $M(P_i)$ is the node to which P_i is assigned for execution. Let $N_{Pi} \subseteq N$ be the set of nodes, to which P_i can be potentially mapped. We know the worst-case execution time (WCET) $C_{P_i}^{N_k}$ of process P_i, when executed on each node $N_k \in N_{Pi}$ [Puschner and Burns, 2000]. Processes mapped on different computation nodes communicate with a message sent over the bus. We consider that the worst-case size of messages is given and we implicitly translate it into the worst-case transmission time on the bus. If processes are mapped on the same node, the message transmission time between them is accounted for in the worst-case execution time of the sending process.

Each computation node N_j has a real-time kernel as its main component. The kernel invokes processes, mapped on N_j, according to the schedule table located in this node. This local schedule table contains all the information that is needed for activation of processes and for sending and receiving communication messages [Pop et al., 2004]. We consider a static bus, which allows sending messages at different times as long as these times are specified in the bus schedule. However, our work can be also used with the static bus that has a limited number of mode changes such as, for example, a TTP bus [Kopetz and Bauer, 2003]. In this case, a frame-packing mechanism can be implemented for incorporating messages [Pop et al., 2005].

In Fig. 2 we have an application A consisting of the process graph G with four processes, P_1, P_2, P_3, and P_4. Processes communicate with messages m_1, m_2, m_3, m_4 and m_5. The deadline is D = 210 ms. The execution times for the processes, if mapped on computation nodes N_1 and N_2, are shown in the table on the right side. “X” in the table



	N_1	N_2
P_1	30	20
P_2	20	X
P_3	X	20
P_4	40	30

Fig. 2. Application Example.

indicates a mapping restriction, i.e., process P_2 can only be mapped on node N_1, and process P_3 can only be mapped on node N_2. The transmission times of messages, if transmitted over the bus, are also indicated in the figure.

4. FAULT TOLERANCE

In this paper we are interested in fault-tolerance techniques for transient faults. 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 computation nodes in the system. Several transient faults may occur simultaneously on several computation nodes as well as several faults may occur on the same computation node. In this paper, we assume that transient faults on the bus are addressed at the communication level, for example, with the use of efficient error correction codes [Piriouet al., 2006; Balakirsky and Vinck, 2006; Emani et al., 2007]and/or through hardware replication of the bus [Kopetz and Bauer, 2003; Silva et al., 2007].

The fault-tolerance mechanisms against transient faults on computation nodes 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 first need to detect the transient faults in order to apply fault tolerance techniques. Error detection can be hardware-based (e.g., watchdogs [Benso et al., 2003], signature checking [Sciuto et al., 1998]) or software-based [Oh et al., 2002a; Nicolescu et al., 2004; Oh et al., 2002b]. We assume that all faults can be found using the above error detection methods. The time needed for detection of faults is accounted for as part of the worst-case execution time (WCET) of the process.

We use process re-execution as a fault tolerance mechanism. The process re-execution operation requires an additional recovery overhead denoted in this paper as μ. The recovery overhead includes the worst-case time that is needed in order to restore process inputs, clean up the node's memory, and re-start process execution. Let us consider the example in Fig. 3, 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. 3, 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 succeed.

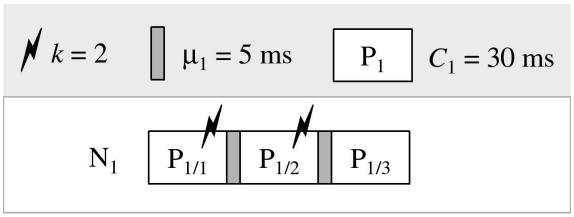


Fig. 3. Re-execution.

5. TRANSPARENCY

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 $A(V, E)$ we will capture the transparency using a function T: $W \rightarrow \{Frozen, Regular\}$, where W is the set of all processes and messages. If $T(w_i) = Frozen$, our scheduling algorithm will handle this transparency requirement (a) by scheduling w_i, if it is a message, at the same transmission time in all alternative execution scenarios and (b) by scheduling the *first* execution instance of w_i, if it is a process, at the same start time in all alternative execution scenarios. In a fully transparent system, all messages and processes are frozen. Systems with a node-level transparency [Kandasamy et al., 2003a; Izosimov et al., 2005; Pop et al., 2009] support a limited transparency setup, in which all the inter-processor messages are frozen while all processes are regular. In such a scheme, system behavior can be observed only by monitoring messages sent to and out of the computation node. It leads to both reduced observability and unnecessary end-to-end delays.

In the example in Fig. 4a, we introduce transparency properties into the application A from Fig. 2. We make process P_3 and messages m_2 and m_3 frozen, i.e., $T(m_2) = Frozen$, $T(m_3) = Frozen$ and $T(P_3) = Frozen$. We will depict frozen processes and messages with squares, while the regular ones are represented by circles. The application has to tolerate $k = 2$ transient faults, and the recovery overhead μ is 5 ms. Processes P_1 and P_2 are mapped on N_1, and P_3 and P_4 are mapped on N_2. Messages m_1, m_2 and m_3 are scheduled on the bus. Four alternative execution scenarios are illustrated in Fig. 4b-e.

The schedule in Fig. 4b corresponds to the fault free scenario. Once a fault occurs in P_4, for example, the scheduler on node N_2 will have to switch to another schedule. In this schedule, P_4 is delayed with $C_4 + \mu$ to account for the fault, where C_4 is the worst-case execution time of process P_4 and μ is the recovery overhead. If, during the second

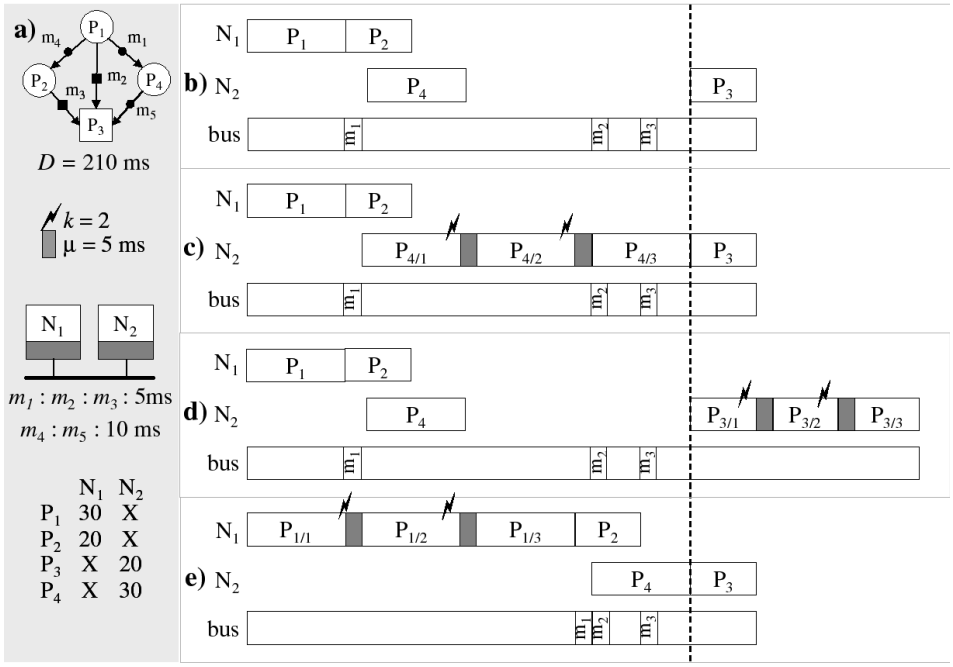


Fig. 4. Application with Transparency.

execution of P_4, a second fault occurs, the scheduler has to switch to another schedule illustrated in Fig. 4c.

Since P_3, m_2 and m_3 are frozen they should be scheduled at the same time in all alternative fault scenarios. For example, re-executions of process P_4 in case of faults in Fig. 4c must not affect the start time of process P_3. The first instance of process P_3 has to be always scheduled at the same latest start time in all execution scenarios, as illustrated with a dashed line crossing Fig. 4. Even if no faults happen in process P_4, in the execution scenarios depicted in Fig. 4d and Fig. 4b, process P_3 will have to be delayed. It leads to a worst-case schedule as in Fig. 4d. Similarly, idle times are introduced before messages m_2 and m_3, such that possible re-executions of processes P_1 and P_2 do not affect the sending times of these messages. Message m_1, however, will be sent at different times depending on fault occurrences in P_1, as illustrated in Fig. 4e.

In Section 5.1, we further illustrate the transparency/performance trade-offs. In Section 5.2, we will discuss the importance of considering transparency properties during mapping.

5.1 Transparency/Performance Trade-offs

In Fig. 5 we illustrate three alternatives, representing different transparency/performance setups for the application A in Fig. 2. The entire system has to tolerate k = 2 transient faults in the hyperperiod of the application, and the recovery overhead μ is 5 ms. Processes P_1 and P_2 are mapped on N_1, and P_3 and P_4 are mapped on N_2. For each transparency alternative (a–c), we show the schedule when no faults occur (a_1–c_1) and also depict the worst-case scenario, resulting in the longest schedule (a_2–c_2). The end-to-

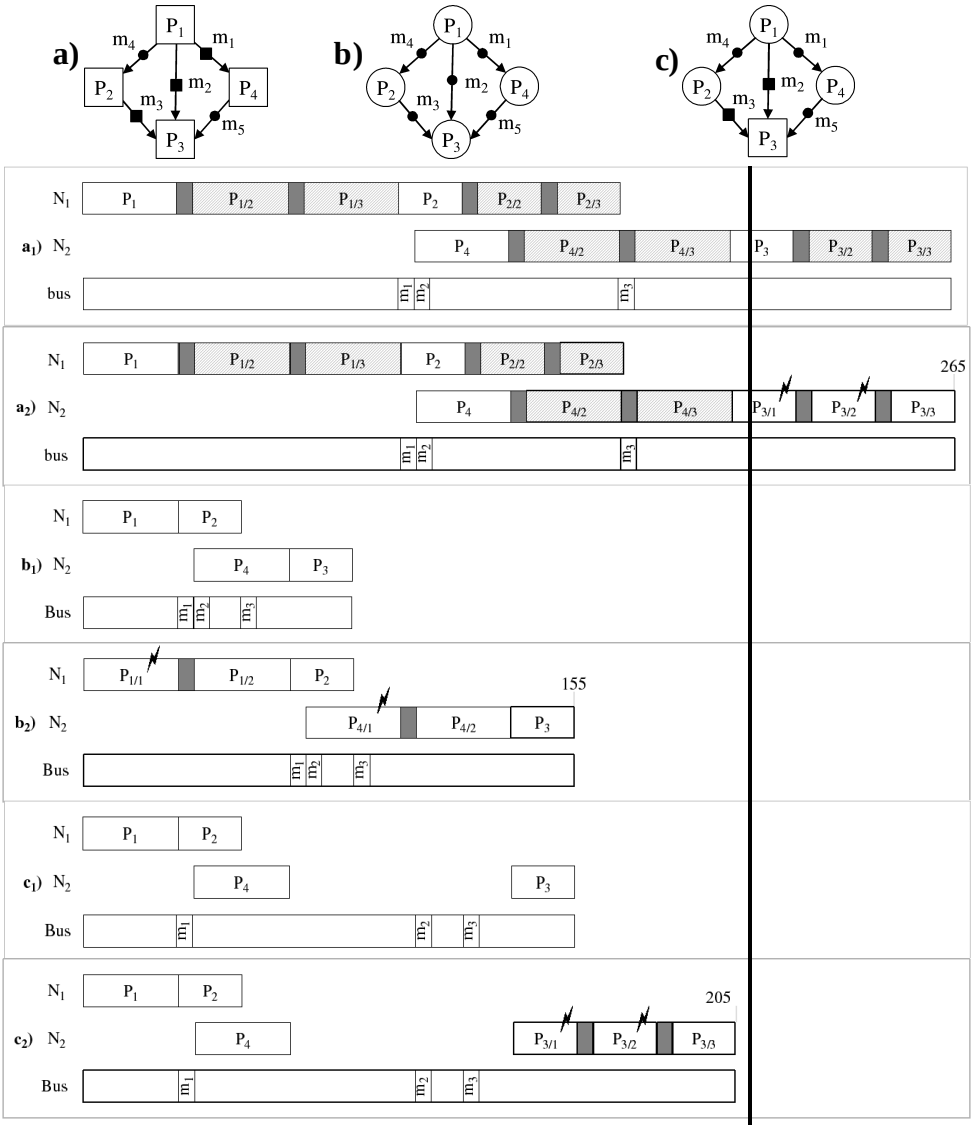


Fig. 5. Trade-off between Transparency and Performance

end worst-case delay of an application will be given by the maximum finishing time of any alternative schedule. Thus, we would like the worst-case schedules in Fig. $5a_2$-c_2 to meet the deadline of 210 ms depicted with a thick vertical line.

In Fig. $5a_1$ and $5a_2$ we show a schedule produced with a *fully transparent* alternative, in which all processes and messages are frozen. We can observe that processes and messages are scheduled at the same time, indifferent of the actual occurrence of faults. The shaded slots in the schedules indicate the intervals reserved for re-executions that are needed to recover from fault occurrences. In general, a *fully transparent* approach, as depicted in Fig. $5a_1$ and Fig. $5a_2$, has the drawback of producing long schedules due to complete lack of flexibility. The worst-case end-to-end delay in the case of full transparency, for this example, is 265 ms, which means that the deadline is missed.

The alternative in Fig. 5b does not have any transparency restrictions. Fig. $5b_1$ shows the execution scenario if no fault occurs, while $5b_2$ illustrates the worst-case scenario. In the case without frozen processes/messages, a fault occurrence in a process P_i can affect the schedule of another process P_j. This allows to build schedules customized to the actual fault scenarios and, thus, are more efficient. In Fig. $5b_2$, for example, a fault occurrence in P_1 on N_1 will cause another node N_2 to switch to an alternative schedule that delays the activation of P_4. P_4 receives message m_1 from P_1. This would lead to a worst-case end-to-end delay of only 155 ms, as depicted in Fig. $5b_2$, that meets the deadline.

However, transparency could be highly desirable and a designer would like to introduce transparency at certain points of the application without violating the timing constraints. In Fig. 5c, we show a setup with a fine-grained, customized transparency, where process P_3 and its input messages m_2 and m_3 are frozen. In this case, the worst-case end-to-end delay of the application is 205 ms, as depicted in Fig. $5c_2$, and the deadline is still met.

5.2 Mapping with Transparency Constraints

In Fig. 6 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 figure. We impose a deadline of 155 ms for the application.

If we do not impose any transparency requirements, i.e., all processes and messages are regular, the optimal mapping is the following: processes P_2, P_4 and P_5 are mapped on

node N_1, while P_1, P_3 and P_6 on node N_2. For this mapping, Fig. 6a$_1$ shows the non-fault scenario, while Fig. 6a$_2$ depicts the worst-case scenario. As observed, the application is schedulable.

If the same mapping determined in Fig. 6a is used with considering process P_2 frozen, the worst-case scenario is depicted in Fig. 6b$_1$. In order to satisfy transparency (start time of P_2 identical in all scenarios) the start time of P_2 is delayed according to the worst-case finishing time of P_4. Thus, the deadline is violated. We will improve the schedule for the worst-case scenario if process P_2 is inserted between re-executions of process P_4, as

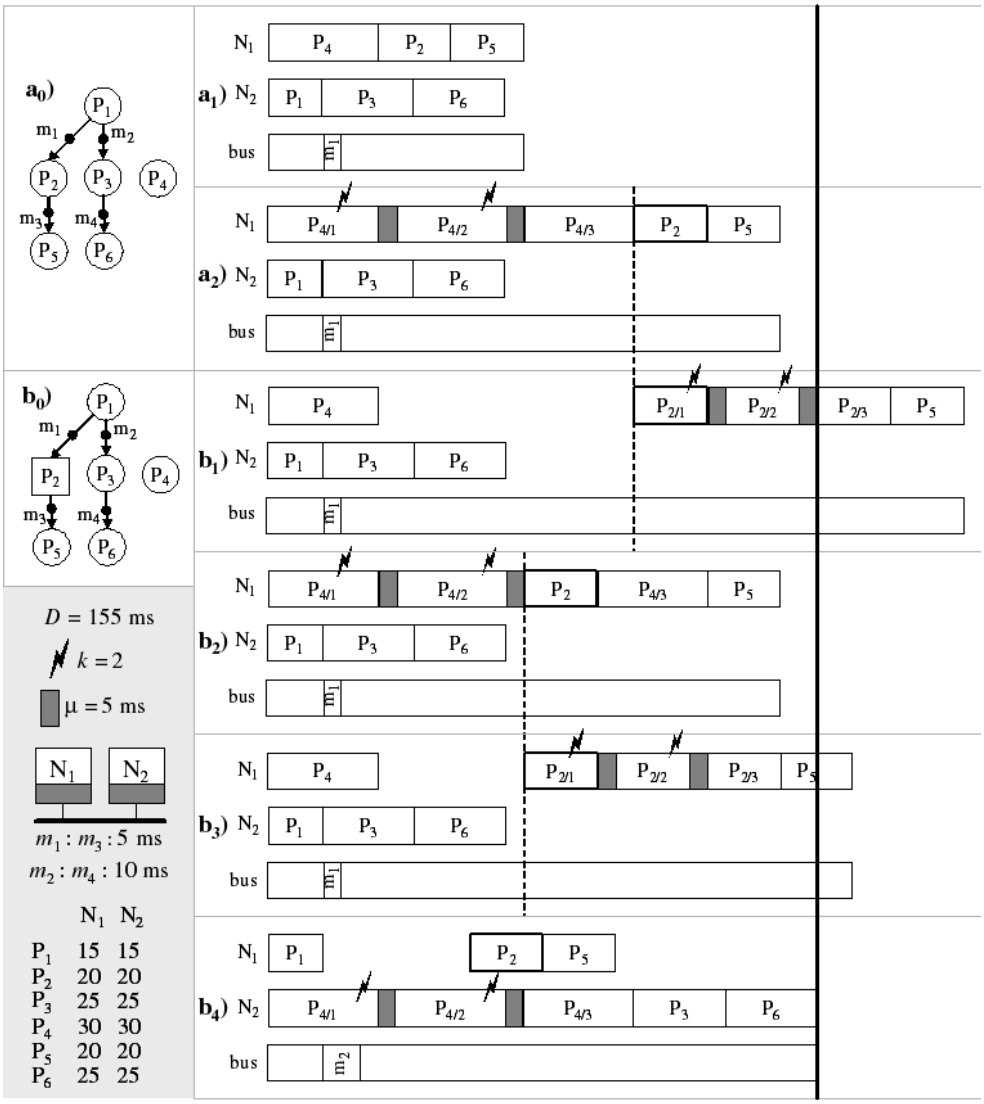


Fig. 6. Mapping and Transparency.

shown in Fig. 6b$_2$. In this case, process P_2 will always start at 70 ms. Even though the worst-case scenario depicted in Fig. 6b$_3$ is better than the one in Fig. 6b$_1$, the deadline is, as before, violated. Only another mapping will make the system schedulable with a frozen P_2: processes P_1, P_2 and P_5 are mapped on node N_1, while processes P_3, P_4 and P_6 are mapped on node N_2. The worst-case scenario, according to this mapping, is depicted in Fig. 6b$_4$. Counterintuitively, this mapping is less balanced and the amount of communications is increased compared to the previous solution, since we send message m_2 that is two times larger than m_1. Nevertheless, in this case the deadline is satisfied. This illustrates that a mapping optimal for an unrestricted design is unsuitable if transparency is imposed.

6. FAULT-TOLERANT SCHEDULING AND DESIGN OPTIMIZATION

In this section, we formulate the design problem and present our design optimization strategy.

6.1 Problem Formulation

As an input, we get a virtual application A, composed from a set of applications A_k (see Section 3). Application A runs on a bus-based architecture consisting of a set of hardware nodes N interconnected via a broadcast bus B. The transparency requirements T on the application, the deadlines, the maximum number k of transient faults, and the recovery overhead μ are given. We know the worst-case execution times for each process on each computation node. The maximum transmission time for all messages, if sent over the bus B, is given.

As an output, we have to produce (1) the mapping of the processes to the computation nodes and (2) the fault-tolerant schedule S, such that maximum k transient faults are tolerated by re-execution, the transparency requirements are considered, and deadlines are satisfied even in the worst-case fault scenario.

6.2 Overall Strategy

The design problem outlined above is NP complete [Ullman, 1975] and is, therefore, addressed using heuristics. In our strategy, illustrated in Fig. 7, we start by determining an initial mapping M_{init} with the InitialMapping function (line 1). This is a straightforward mapping that balances computation node utilization and minimizes communications. The schedulability of the resulted system is evaluated with the conditional scheduling algorithm (lines 2–3) from Section 7.3. 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 mapping heuristic that combines a greedy algorithm and a method to recover from local optima.

A new mapping alternative M_{new} is obtained with a greedy algorithm, IterativeMapping (line 5), presented in Section 6.3. The algorithm uses as a cost function the schedule length estimated with the heuristic presented in Section 7.4, in order to evaluate the intermediate mapping decisions. This approach, where we use estimation instead of the actual scheduling, reduces the runtime and speeds-up the optimization. However, the final mapping solution obtained with the IterativeMapping has to be evaluated with the actual conditional scheduling algorithm (line 6) from Section 7.3. The scheduling algorithm will produce the schedule tables and will determine exactly if the application with the proposed mapping is schedulable.

Since IterativeMapping is a greedy heuristic it will very likely end up in a local minimum M_{new}. If M_{new} is not schedulable, in order to explore other areas of the design space, we will restart the IterativeMapping heuristic with a new initial solution M_{init}. This

```
OptimizationStrategy(G, T, k, N, B, D)
M_init = InitialMapping(G, N, B)
S = FTScheduleSynthesis(G, T, k, N, B, M_init)
if deadlines are met then return M_init
while not_termination do
  M_new = IterativeMapping(G, T, k, N, B, M)
  S = FTScheduleSynthesis(G, T, k, N, B, M_new)
  if deadlines are met then return {M_new, S}
  M_init = FindNewInit(G, N, B, M_new)
end while
return no_solution
end OptimizationStrategy
```

Fig. 7. Optimization Strategy.

solution is constructed such that it will reduce the likelihood of ending in the same local minimum again. As recommended in literature [Reevs, 1993], we perform a diversification of the current solution by running another mapping optimization with a cost function different from the "goal" cost function of the IterativeMapping algorithm. This optimization will produce a new mapping M_{init} and is implemented by the function FindNewInit (line 8). This function runs a simple greedy iterative mapping, which, instead of the schedule length, 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 have obtained empirically, is to limit the number of consecutive iterations without any improvement of the schedule length to $N_{proc} \times k \times ln(N_{compnodes})$, where N_{proc} is the number of processes, $N_{compnodes}$ is the number of computation nodes, and k is the maximum number of faults in the system period. An increase in any of these parameters would contribute to the increase of the design space exploited by our mapping heuristic. In particular, the number of computation nodes contributes to the most significant increase in the design space. Thus, we use $ln(N_{compnodes})$ to capture this issue. Note that the design space does not grow linearly with the increase of these parameters, i.e., it grows exponentially. However, formula $N_{proc} \times k \times ln(N_{compnodes})$ allows us to efficiently capture this growth, yet without dramatic increase in the execution time of the algorithm.

6.3 Iterative Mapping Heuristic

Our mapping algorithm, IterativeMapping, depicted in Fig. 8 is a greedy algorithm that incrementally changes the mapping M until no further improvement (line 3) is produced. Our approach is to tentatively change the mapping of processes on the critical path of the application graph G. The critical path CP is found by the function FindCP (line 6). Each process P_i in the list CP is then tentatively moved to each node in N. We evaluate each move in terms of schedule length, considering transparency properties T and the number of faults k (line 10).

The calculation of the schedule length should, in principle, be performed by conditional scheduling (FTScheduleSynthesis function, see Section 7.3). However, conditional 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, presented in Section 7.4. This heuristic is used to guide the IterativeMapping algorithm.

```
IterativeMapping(G, T, k, N, B, M)
improvement := true
I_best := ScheduleLengthEstimation(G, T, k, N, B, M)
while improvement do
  improvement := false
  P_best := ; N_best :=
  CP := FindCP(G)
  for processes P_i ∈ CP do
    for each N_j  N_c do
      ChangeMapping(M, P_i, N_j)
      I_new := ScheduleLengthEstimation(G, T, k, N, B, M)
      RestoreMapping(M)
      if I_new < I_best then
        P_best := P_i; N_best := N_j; I_best := I_new
        improvement := true
      end if
    end for
  end for
  if improvement then ChangeMapping(M, P_best, N_best)
end while
return M
end IterativeMapping
```

Fig. 8. Iterative Mapping Heuristic (IMH).

After evaluating possible alternatives, the best move consisting of the best process P_{best} and the best target computation node N_{best} is selected (lines 12–15). This move is executed if leading to an improvement (line 18). IterativeMapping will stop if there is no further improvement.

The final solution produced with the IterativeMapping heuristic will have to be evaluated with the conditional scheduling algorithm, as discussed in the previous section (see line 6 in Fig. 7). If the final solution is valid according to this evaluation, we will conclude that the system is schedulable. If the solution is not valid, IterativeMapping will be run once again after performing diversification of the obtained mapping solution with the FindNewInit function (line 8, Fig. 7).

7. CONDITIONAL SCHEDULING

Our conditional scheduling technique is based on the fault-tolerant process graph (FTPG) representation and generates, as output, a set of schedule tables. Schedule tables, discussed in Section 7.2, are used by a distributed run time scheduler for executing processes on the computation nodes.

7.1 Fault-Tolerant Process Graph

The scheduling technique presented in this section is based on the fault-tolerant process graph (FTPG) representation. FTPG captures alternative schedules in the case of different fault scenarios. Every possible fault occurrence is considered as a condition that is "true" if the fault happens and "false" if the fault does not happen. FTPG allows to efficiently and correctly model recovery in the context of multiple transient faults, which is utilized by our conditional scheduling algorithm.

In Fig. 9a we have an application *A* modelled as a process graph *G*. The application *A* can experience at most two transient faults (for example, one during the execution of process P_2, and one during P_4, as illustrated in the figure). 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 *A* to a fault-tolerant process graph (FTPG) *G*, represented in Fig. 9b. In an FTPG the fault occurrence information is represented as *conditional edges* and the frozen processes/ messages are captured using *synchronization nodes*. One of the conditional edges, for

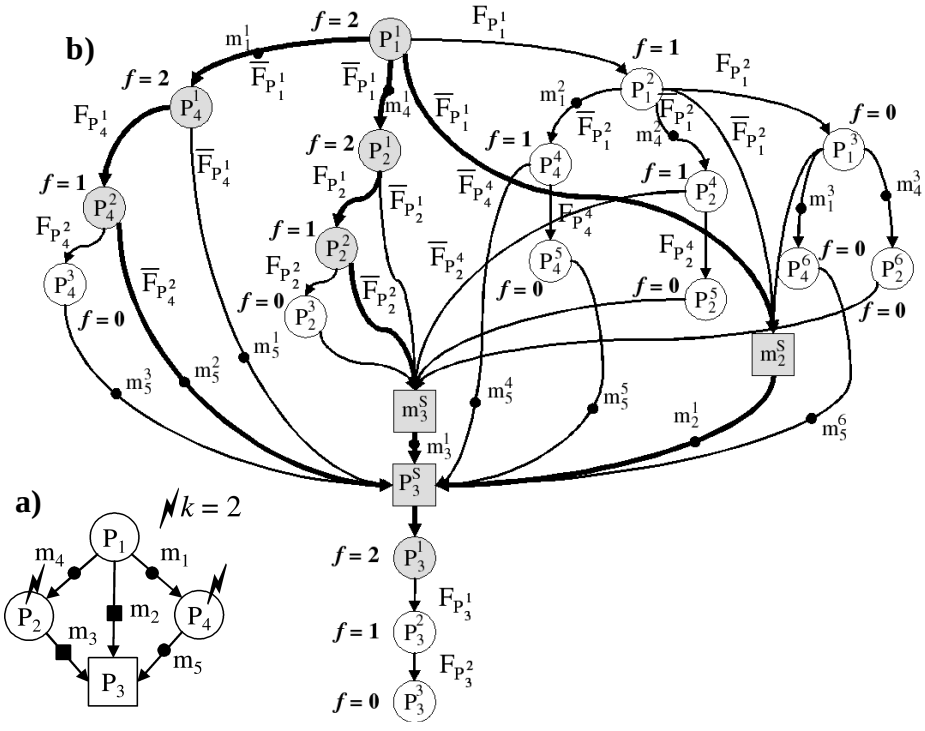


Fig. 9. Fault-Tolerant Process Graph.

example, is P_1^1 to P_4^1 in Fig. 9b, 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 FTPG in Fig. 9b captures all the fault scenarios that can happen during the execution of application *A* in Fig. 9a. The subgraph marked with thicker edges and shaded nodes in Fig. 9b captures the execution scenario when processes P_2 and P_4 experience one fault each. We will refer to every such subgraph corresponding to a particular execution scenario as an *alternative trace* of the FTPG. The fault occurrence possibilities for a given process execution, for example P_2^1, the first execution of P_2, are captured by the conditional edges $F_{P_2^1}$ (fault) and $\overline{F}_{P_2^1}$ (no-fault). The transparency requirement that, for example, P_3 has to be frozen, is captured by the synchronization node P_3^S, which is inserted, as shown in Fig. 9b, before the copies corresponding to the possible executions of process P_3. The first execution copy P_3^1 of process P_3 has to be immediately scheduled after its synchronization node P_3^S. In Fig. 9b, process P_1^1 is a conditional process because it "produces" condition $F_{P_1^1}$, while P_1^3 is a regular process. In the same figure, m_2^S and m_3^S, similarly to P_3^S, are synchronization nodes (depicted with a rectangle). Messages m_2 and m_3 (represented with their single copies m_2^1 and m_3^1 in the FTPG) have to be immediately scheduled after synchronization nodes m_2^S and m_3^S, respectively.

Regular and conditional processes are activated when all their inputs have arrived. A synchronization node, however, is activated after inputs coming on one of the alternative paths, corresponding to a particular fault scenario, have arrived. For example, a transmission on the edge $e_{12}^{1S_m}$, labeled $\overline{F}_{P_1^1}$, will be enough to activate m_2^S.

A guard is associated to each node in the graph. An example of a guard associated to a node is, for example, $K_{P_2^2} = \overline{F}_{P_1^1} \wedge F_{P_2^1}$, indicating that P_2^2 will be activated in the fault scenario where P_2 will experience a fault, while P_1 will not. A node is activated only in a scenario corresponding to which the value of the associated guard is true.

Definition. Formally, an FTPG corresponding to an application $A = G(V, E)$ is a directed acyclic graph $G(V_P \cup V_C \cup V_T, E_S \cup E_C)$. We will denote a node in the FTPG with that will correspond to the m^{th} copy of process $P_i \in V$. Each 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.

Each node $v_i \in V_T$ is a *synchronization node* and represents the synchronization point corresponding to a frozen process or message. We denote with P_i^S the synchronization node corresponding to process $P_i \in A$ and with m_i^S the synchronization node corresponding to message $m_i \in A$. Synchronization nodes will 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:

- $e_{ij}^{mS} \in E_S$ from P_i^m to P_j^S;
- $e_{ij}^{Sn} \in E_S$ from P_i^S to P_j^n;
- $e_{ij}^{mS_m} \in E_S$ from P_i^m to m_j^S;
- $e_{ij}^{S_m n} \in E_S$ from m_i^S to P_j^n;
- $e_{ij}^{SS} \in E_S$ from P_i^S to P_j^S;
- $e_{ij}^{S_m S} \in E_S$ from m_i^S to P_j^S;
- $e_{ij}^{SS_m} \in E_S$ from P_i^S to m_j^S; and
- $e_{ij}^{S_m S_m} \in E_S$ from m_i^S to m_j^S.

Edges $e_{ij}^{mn} \in E_C$, $e_{ij}^{mS} \in E_C$, and $e_{ij}^{mS_m} \in E_C$ 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[2]. Note that edges e_{ij}^{Sn}, $e_{ij}^{S_m n}$, e_{ij}^{SS}, $e_{ij}^{S_m S}$, $e_{ij}^{SS_m}$, and $e_{ij}^{S_m S_m}$ coming from a synchronization node cannot be conditional.

A boolean expression $K_{P_i^m}$, called guard, is associated to each node P_i^m in the graph. The guard captures the necessary activation conditions (fault scenario) for the respective node.[3]

7.2 Schedule Table

The output produced by the FTPG scheduling algorithm that will be discussed in the next section is a schedule table that contains all the information needed for a distributed run time scheduler to take decisions on activation of processes and sending of messages. 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 fault occurrences.

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, i.e., decisions related to processes located on the respective nodes. Fig. 10 presents the schedules for nodes N_1 and N_2, which will be produced by the conditional scheduling algorithm in Fig. 11 for the FTPG in Fig. 9. Processes P_1 and P_2 are mapped on node N_1, while P_3 and P_4 on node N_2.

In each table there is one row for each process and message from application *A*. A row contains activation times corresponding to different guards, or *known conditional values*, that are depicted as a conjunction in the head of each column in the table. A particular conditional value in the conjunction indicates either a success or a failure of a certain process execution. The value, “true” or “false”, respectively, is produced at the end of each process execution (re-execution) and is immediately *known* to the computation node on which this process has been executed. However, this conditional value is *not yet known* to the other computation nodes. Thus, the conditional value generated on one computation node has to be *broadcasted* to the other computation nodes, encapsulated in a *signalling message*. Signalling messages have to be sent at the earliest possible time since the conditional values are used to take the best possible decisions on process activation [Eles et al., 2000]. Only when the condition is known,

[2]They can only meet in a synchronization node.

[3]We present the algorithm for FTPG generation in Appendix I.

i.e., has arrived with a signalling message, a decision will be taken that depends on this condition. In the schedule table, there is one row for each signalling message with the condition whose value has to be broadcasted to other computation nodes.

According to the schedule for node N_1 in Fig. 10a, 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 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 (no fault in P_1), at $t = 100$ if $F_{P_1^1} \wedge F_{P_1^2}$ is true (faults in P_1 and its first re-execution), etc.

To produce a deterministic behavior 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 activation, 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 determined at the respective moment t and are *known* to the processing element that executes P_i.

a)

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									

b)

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)				

Fig. 10. Conditional Schedule Tables.

7.3 Scheduling Algorithm

According to our FTPG model, some processes will only be activated if certain conditions (i.e., fault occurrences), produced by previously executed processes, are fulfilled. Thus, 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 regarding which process to activate and at which time has to be taken without knowing which values some of the conditions will later get. On the other hand, 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 [Ullman, 1975] in even simpler contexts than that of FTPG scheduling. 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. 11. The FTScheduleSynthesis function produces the schedule table *S*, while taking as input the application graph *G* with the transparency requirements *T*, the maximum number *k* of transient faults that have to be tolerated, the architecture consisting of computation nodes *N* and bus *B*, and the mapping *M*.

```
FTScheduleSynthesis(G, T, k, N, B, M)
S = ∅; G = BuildFTPG(G, T, k)
L_S = GetSynchronizationNodes(G)
PCPPriorityFunction(G, L_S)
if L_S ≡ ∅ then
  FTPGScheduling(G, M, ∅, S)
else
  for each S_i ∈ L_S do
    t_max = 0; K_Si = ∅
    {t_max, K_Si} = FTPGScheduling(G, M, S_i, S)
    for each K_j ∈ K_Si do
      Insert(S, S_i, t_max, K_j)
    end for
  end for
end if
return S
end FTScheduleSynthesis
```

Fig. 11. Fault-Tolerant Schedule Synthesis Strategy.

Our synthesis approach employs a *list scheduling* based heuristic, FTPGScheduling, presented in Fig. 13, 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 7.2, 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 (Fig. 11), we initialize the schedule table S and build the FTPG G as presented in Appendix I.[4] If the FTPG does not contain any synchronization node ($L_s \equiv \varnothing$), we perform the FTPG scheduling for the whole FTPG graph at once (lines 4–5).

If the FTPG contains at least one synchronization node $S_i \in L_S$, the procedure is different (lines 7–13). A synchronization node S_i must have the same start time t_i in the schedule S, regardless of the guard K_{Si}. The guard captures the necessary activation conditions for S_i under which it is scheduled. For example, the synchronization node m_2^S in Fig. 12 has the same start time of 105, in each corresponding column of the table in Fig. 10. In order to determine the start time t_i of a synchronization node $S_i \in L_S$, where L_S is the list of synchronization nodes, we will have to investigate all the alternative fault-scenarios (modelled as different alternative paths through the FTPG) that lead to S_i. Fig. 12 depicts the three alternative paths that lead to m_2^S for the graph in Fig. 9b. These paths are generated using the FTPGScheduling function (called in line 9, Fig. 11). This function records the maximum start time t_{max} of S_i over the start times in all the alternative paths. In addition, FTPGScheduling also records the guards K_{Si} 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 K_{Si} at the unique time t_{max} (line 11 in Fig. 11). For example, m_2^S is inserted at time $t_{max} = 105$ in the columns corresponding to $Km_2 = \{ \overline{F}_{P_1^1}, F_{P_1^1} \wedge F_{P_1^2}, F_{P_1^1} \wedge \overline{F}_{P_1^2} \}$.

The FTPGScheduling function is based on list scheduling and it calls itself for each conditional process in the FTPG G in order to separately schedule the *faulty* branch and the *no fault* branch (lines 21 and 23, Fig. 13). Thus, the alternative paths are not activated simultaneously and resource sharing is correctly achieved. Signalling messages,

[4]For efficiency reasons, the actual implementation is slightly different from the one presented here. In particular, the FTPG is not explicitly generated as a preliminary step of the scheduling algorithm. Instead, during the scheduling process, the currently used nodes of the FTPG are generated on the fly.

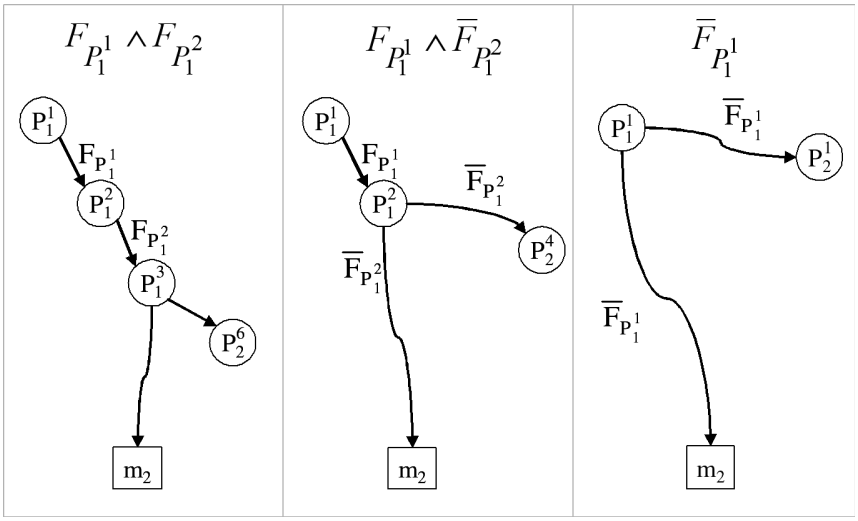


Fig. 12. Alternative paths investigated by FTPGScheduling for the synchronization node m_2^S.

.

transporting condition values, are scheduled (line 19), and only when the signalling message arrives to the respective computation node, the scheduling algorithm can account for the received condition value and activate processes and messages, associated with this computation node on the corresponding conditional branch of the FTPG.

List scheduling heuristics use priority lists from which ready nodes (vertices) in an application graph are extracted in order to be scheduled at certain moments. A node in the graph is "ready" if all its predecessors have been scheduled. Thus, in FTPGScheduling, for each resource $R_j \in R$, where the set R of resources contains all the computation nodes $N_i \in N$ and the bus B, the *highest priority ready node* X_i is extracted from the head of the local priority list L_{Rj} (line 3). We use the *partial critical path* (PCP) priority function [Eles et al., 2000] in order to assign priorities to the nodes (line 3 in FTScheduleSynthesis, Fig. 11).

X_i can be a synchronization node, a copy of a process, or a copy of a message in the FTPG G. If the ready node X_i is the currently investigated synchronization node S (line 8), the latest start time and the current guards are recorded (lines 10–11). If other unscheduled synchronization nodes are encountered, they will not be scheduled yet (lines 14–15), since FTPGScheduling investigates one synchronization node at a time. Otherwise, i.e., if not a synchronization node, the current ready node X_i is placed in the schedule S at time t under guard K.[5] The time t is the time when the resource R_j is

[5]Recall that synchronization nodes are inserted into the schedule table by the FTScheduleSynthesis function on line 11 in Fig. 11.

```
FTPGScheduling(G, M, S, S)
while                          do
  for each R_j ∈ N ∪ {B} do
    L_Rj = LocalReadyList(S, R_j, M ) -- find unscheduled ready nodes on resource R_j
    while L_Rj ≠ ∅ do
      X_i := Head (L_Rj)
      t = ResourceAvailable(R_j, X_i) -- the earliest time when R_j can accommodate X_i
      K = KnownConditions(R_j, t) -- the conditions known to R_j at time t
      if X_i ≡ S then -- synchronization node currently under investigation
          if t > t_max then
            t_max = t -- the latest start time is recorded
            K_Si = K_Si ∪ {K} -- the guard of the synchronization node is recorded
          end if
          return {t_max, K_Si} -- exploration stops at the synchronization node S
      else if X_i ∈ V_T and X_i is unscheduled then -- other synchronization nodes
          continue -- are not scheduled at the moment
      end if
      Insert(S, X_i, t, K) -- the ready node X_i is placed in the schedule S under guard K
      if X_i ∈ V_C then -- conditional process
          Insert(S, SignallingMsg(X_i), t, K) -- broadcast conditional value
          -- schedule the faulty branch
          FTPGScheduling(G, L_Rj ∪ GetReadyNodes(X_i, true)) --recursive call for true branch
          -- schedule the non-faulty branch
          FTPGScheduling(G, L_Rj ∪ GetReadyNodes(X_i, false)) --recursive call for false branch
      else
          L_Rj = L_Rj ∪ GetReadyNodes(X_i)
      end if
    end while
  end for
end while
end FTPGScheduling
```

Fig. 13. Conditional Scheduling.

available (line 17). Guard K on the resource R_j is determined by the KnownConditions function (line 7). Our approach eliminates from K those conditions that, although known to R_j at time t, will not influence the execution of X_i. For example, frozen processes and messages are not influenced by any condition.

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 times on the resources. Thus, our ResourceAvailable function (line 6, Fig. 13) will determine the start time $t \geq t_{asap}$ in the first *continuous* segment of time, available on resource R_j, large enough to accommodate X_i, if X_i is scheduled at this start time t. t_{asap} is the earliest possible start time of X_i in the considered execution scenario. For example, as outlined in the schedule table in Fig. 10a, 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 31, 100 and 66, within this idle segment (see Fig. 10a).

7.4 Schedule Length Estimation

The worst-case fault scenario consists of a combination of k fault occurrences that leads to the longest schedule. The conditional scheduling algorithm, presented in Section 7.3, examines all fault scenarios captured by the fault-tolerant process graph (FTPG), produces the fault-tolerant schedule table, and implicitly determines the worst-case fault scenario.

However, the number of alternative paths to investigate is growing exponentially with the number of faults. Hence, conditional scheduling is too slow to be used inside the iterative mapping loop discussed in Section 6.3 (Fig. 8). On the other hand, mapping optimization does not require generation of complete schedule tables. Instead, only an estimation of the schedule length is needed in order to evaluate the quality of the current design solution. Hence, in this section, we present a worst case schedule length estimation heuristic.

The main idea of our estimation is to avoid investigating all fault scenarios since it is time-consuming. Instead, the estimation heuristic incrementally builds a fault scenario that is as close as possible (in terms of the resulted schedule length) to the worst case.[6]

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 FTPG $G_X(m)$ that 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 $P_i \in G_X(m)$, which introduces the largest delay on the critical path if it experiences the $(m+1)^{th}$ 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. 14a. The set L_S is prepared, by extracting all synchronization nodes from the application graph G (line 2). If the application A does not contain frozen processes and messages, i.e., $L_S \equiv \varnothing$, we will

[6]The schedule length estimation in the present context is not required to be safe (pessimistic) because, in order to guarantee schedulability, we apply the actual conditional scheduling algorithm after the mapping has been obtained (see OptimizationStrategy in Fig. 7, line 6).

a) **ScheduleLengthEstimation(**G , T , k, N , B, M**)**

ψ = LastNode(G)
L_S = ExtractSynchronizationNodes(G)
if $L_S \neq \emptyset$ **then**
  PCPPriorityFunctionSort(G, L_S)
  for each $S_i \in L_S$ **do**
    $t_{max\_start}\{S_i\}$ = StartTimeEstimation(G , T , k, N , B, M , S_i)
    FixStartTime(S_i, G , t_{max_start})
  end for
end if
$t_{max\_start}\{\psi\}$ = StartTimeEstimation(G , T, k, N , B, M , ψ)
$SL_{max} = t_{max\_start}\{\psi\} + worst\_exec\_time(\psi)$
return SL_{max}
end ScheduleLengthEstimation

b) **StartTimeEstimation(**G , T , k, N , B, M , *Target*)

$t_{max\_start} = 0$; $X(0) = \emptyset$
Z = SelectProcesses(*Node*, G)
for $m = 1...k$ **do**
  for each $P_i \in Z$ **do**
    $G_{X(m),\,i}$ = CreatePartialFTCPG($X(m-1)$, P_i)
    t_i = ListScheduling($G_{X(m),\,i}$, *Target*)
    if $t_{max\_start} < t_i$ **then**
      $t_{max\_start} = t_i$
      $P_{worst} = P_i$
    end if
  end for
  $X(m) = X(m-1) + P_{worst}$
end for
return t_{max_start}
end StartTimeEstimation

Fig. 14. Schedule Length Estimation.

directly estimate the latest start time of the last process ψ in the graph G with the StartTimeEstimation heuristic (line 10).

The StartTimeEstimation heuristic is outlined in Fig. 14b. It implements our idea for fast estimation discussed above. StartTimeEstimation receives as an input a *Target* node, the start time of which has to be estimated. At first, StartTimeEstimation selects a set Z of processes, whose re-executions will potentially introduce the largest delays to the start time of *Target* (line 2). These re-executions will be considered for generation of partial FTPGs $G_X(m)$, increasing the number of faults m from 1 to k (lines 3-5). Each m-fault scenario in partial FTPG $G_X(m)$, i (the mth fault occurs in process P_i) is evaluated with a

ListScheduling heuristic that stops once it reaches *Target* (line 6).[7] If the obtained start time t_i is larger than the largest-so-far start time t_{max_start}, it is saved as t_{max_start} (line 8). Process P_i is saved as P_{worst}. After evaluation of all selected processes, P_{worst} will contain the process that has led to the latest start time of *Target*. This process will be used in construction of the $m + 1$ faults scenarios (line 12) for the next iterations. Once we reach k faults and evaluate the respective k fault scenarios, the estimation heuristic will return the corresponding t_{max_start} as the latest start time of *Target* (line 14).

If the set L_S in ScheduleLengthEstimation is not empty (line 3, Fig. 14a), i.e., the application A contains at least one frozen process or message, we will estimate latest start times for all synchronization nodes (lines 4-8, Fig. 14a). We will order synchronization nodes according to the PCP priority and will investigate them one-by-one (lines 4-5). In order to obtain the latest start time of each next node S_i, we need to consider the latest start times of all previous synchronization nodes because they will significantly influence the start time of node S_i. Thus, when we obtain the latest start time t_{max_start} for each node S_i, we "fix" this time in the graph G (FixStartTime function, line 7), so that S_i's start time is considered for the next synchronization node. The latest start time of the last process ψ will be estimated considering latest start times of all synchronization nodes in G (line 10).

Finally, the estimated longest schedule length SL_{max} will be obtained starting the last process ψ at its latest start time t_{max_start} (lines 11-12, Fig. 14a).

8. EXPERIMENTAL RESULTS

In order to demonstrate the effectiveness of the proposed algorithms, we run a set of extensive experiments both on synthetic applications and a real-life example. First, we evaluate our conditional scheduling algorithm and compare it to shifting-based scheduling proposed in [Izosimov et al., 2005; Pop et al., 2009]. Then, we study properties of our mapping optimization algorithm that uses as a cost function the schedule length estimation. We also evaluate the estimation in terms of monotonicity, by comparing its results to the results produced with the actual conditional scheduling. Finally, we apply our scheduling and mapping algorithms to a real-life example, a vehicle cruise controller.

[7] ListScheduling is a list scheduling based heuristic with the PCP priority function as in the conditional scheduling algorithm in Section 7.3.

8.1 Scheduling with Fault Tolerance

For the evaluation of our scheduling algorithm we have 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. These faults can happen during one execution cycle. The duration μ of the recovery time has been set to 5 ms. Fifteen examples have been randomly generated for each application dimension, thus a total of 60 applications have been used for experimental evaluation. We have generated both graphs with random structure and graphs based on more regular structures like trees and groups of chains. Execution times and message lengths have been assigned randomly within the interval 10 to 100 ms, and 1 to 4 bytes range, respectively. To evaluate the scheduling, we have first generated a fixed mapping on the computation nodes with our design optimization strategy from [Izosimov et al., 2005; Pop et al., 2009]. The experiments have been run on Sun Fire V250 computers.

We were first interested to evaluate how the conditional scheduling algorithm handles the transparency/performance trade-offs imposed by the designer. Hence, we have scheduled each application, on its corresponding architecture, using the conditional scheduling (CS) strategy from Fig. 11. In order to evaluate CS, we have considered a reference non-fault tolerant implementation, NFT. NFT executes the same scheduling algorithm but considering that no faults occur ($k = 0$). Let δ_{CS} and δ_{NFT} be the end-to-end delays of the application obtained using CS and NFT, respectively. The fault tolerance overhead is defined as $100 \times (\delta_{CS} - \delta_{NFT}) / \delta_{NFT}$.

We have considered five transparency scenarios, depending on how many of the inter-processor messages have been set as frozen: 0, 25, 50, 75 or 100%. Table I presents the average fault-tolerance overheads for each of the five transparency requirements. We see that, as the transparency requirements are relaxed, the fault-tolerance overheads are reduced. Thus, the designer can trade-off between the degree of transparency and the overall performance (schedule length). For example, for application graphs of 60

Table I. Fault-Tolerance Overheads (CS), %.

% Frozen messages	20 processes			40 processes			60 processes			80 processes		
	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 II. Memory Requirements (CS), Kbytes.

% Frozen messages	20 processes			40 processes			60 processes			80 processes		
	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%	0.1	0.3	0.5	0.4	0.9	1.7	0.7	2.1	4.4	1.2	4.2	8.8
75%	0.2	0.6	1.4	0.6	2.1	5.0	1.2	4.6	11.6	2.0	8.4	21.1
50%	0.3	0.8	1.9	0.8	3.1	8.1	1.5	7.1	18.3	2.6	12.2	34.5
25%	0.3	1.2	3.0	1.0	4.3	12.6	1.9	10.0	28.3	3.1	17.3	51.3
0%	0.4	1.4	3.7	1.2	5.6	16.7	2.2	11.7	34.6	3.4	19.3	61.9

processes with three faults, we have obtained an 86% overhead for 100% frozen messages, which is reduced to 58% for 50% frozen messages.

Table II presents the average memory[8] space per computation node (in kilobytes) required to store the schedule tables. Often, one entity has the same start time under different conditions. We merge such entries in the table into a single table entry, headed by the union of the logical expressions. Thus, Table II 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 number of frozen messages from 50% to 100%, reduces the memory needed from 18K to 4K. This demonstrates that transparency can also be used for memory/performance trade-offs.

The CS 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, in the worst case, 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.

Shifting-based scheduling (SBS), proposed in [Izosimov et al., 2005; Pop et al., 2009], always preserves the same order of processes and messages in all execution scenarios and can only handle a very limited setup in which all inter-processor messages are frozen and no other transparency requirements can be captured. As a second set of experiments, we have compared the conditional scheduling approach with the shifting-based scheduling approach. In order to compare the two algorithms, we have determined the end-to-end delay δ_{SBS} of the application when using SBS. For both the SBS and the CS approaches, we have obtained a fixed mapping on the computation nodes with our design optimization strategy from [Izosimov et al., 2005; Pop et al., 2009]. We have considered that all inter-processor messages and only them are frozen. When comparing

[8]Considering an architecture where an *integer* and a *pointer* are represented on two bytes.

the delay δ_{CS}, obtained with conditional scheduling, to δ_{SBS} in the case of, for example, k = 2, conditional scheduling outperforms SBS on average with 13%, 11%, 17%, and 12% for application dimensions of 20, 40, 60 and 80 processes, respectively.

8.2 Mapping Heuristic

For the evaluation of our mapping optimization strategy we have used applications of 20, 30, and 40 processes implemented on an architecture of 4 computation nodes. We have varied the number of faults from 2 to 4 within one execution cycle. The recovery overhead μ has been set to 5 ms. Thirty examples have been randomly generated for each dimension. Execution times and message lengths have been assigned randomly using uniform distribution within the interval 10 to 100 ms, and 1 to 4 bytes, respectively. We have selected a transparency level with 25% frozen processes and 50% frozen inter-processor messages. The experiments have been done on a Pentium 4 processor at 2.8 GHz with 1 Gb of memory.

We were first interested to evaluate the proposed heuristic for schedule length estimation (ScheduleLengthEstimation in Fig. 14, denoted with SE), in terms of monotonicity, relative to the FTScheduleSynthesis (CS) algorithm presented in Section 7.3. SE is monotonous with respect to CS if for two alternative mapping solutions M_1 and M_2 it is true that if $CS(M_1) \leq CS(M_2)$ then also $SE(M_1) \leq SE(M_2)$. This property is important because, with a high monotonicity, the mapping optimization guided by the estimation will follow the same trajectory as it would follow if guided by the actual conditional scheduling.

For the purpose of evaluating the monotonicity of SE with regard to CS, 50 random mapping changes have been performed for each application. Each of those changes has been evaluated with both SE and CS. The results are depicted in Table III. As we see, in over 90% of the cases, SE correctly evaluates the mapping decisions, i.e. in the same way as CS. The monotonicity decreases slightly with the application dimension.

Table III. Monotonicity of Estimation (%).

Number of processes	2 faults	3 faults	4 faults
20	94.20	90.58	91.65
30	89.54	88.90	91.48
40	88.91	86.93	86.32

Table IV. Execution Time (sec): Estimation vs. Scheduling,

Number of processes	2 faults		3 faults		4 faults	
	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

Another important property of SE is its execution time, presented in Table IV. The 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 significantly smaller than that with CS. This shows that the schedule length estimation heuristic is well-suited to be used inside a design space exploration loop.

We were also interested to evaluate our mapping optimization strategy, for the selected transparency level with 25% frozen processes and 50% frozen inter-processor messages. We have compared our mapping optimization that considers fault tolerance with transparency to the mapping optimization strategy proposed in [Izosimov et al., 2005; Pop et al., 2009]that does not consider the transparency/performance trade-offs. For the sake of this comparison, we will refer to our OptimizationStrategy (in Fig. 7) as AWARE and to the latter one as BLIND. In Table V, we show the improvement of AWARE over BLIND in terms of the schedule length corresponding to the produced mapping solution. The schedule length obtained with AWARE is 30% shorter on average. This confirms that considering the transparency properties leads to significantly better design solutions and that the SE heuristic can be successfully used inside an optimization loop. Note that, while SE has been used inside the loop, the final evaluation of the AWARE solutions (as well as of those produced with BLIND) has been done generating the actual schedules with CS.

We were also interested to compare the solutions obtained with AWARE using SE with the case where CS is used for evaluating the mapping alternatives during optimization. However, due to the long optimization times with the CS based exploration, we have run this experiment only for applications of 20 processes. We have chosen 15 synthetic applications with 25% frozen processes and 50% frozen messages. In terms of

Table V. Mapping Improvement (%).

Number of processes	2 faults	3 faults	4 faults
20	32.89	32.20	30.56
30	35.62	31.68	30.58
40	28.88	28.11	28.03

schedule length, in case of 2 faults, the CS-based strategy is only 3.18% better than the SE-based one. In case of 3 faults, the difference is 9.72%, while for 4 faults the difference in terms of obtained schedule length is of 8.94%.

8.3 Real-life Example

We have also used a real-life example implementing a vehicle cruise controller (CC) for the evaluation of our scheduling and mapping algorithms. The process graph that models the CC has 32 processes, and is described in [Izosimov, 2009]. The hardware architecture consists 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.

For the evaluation of the proposed scheduling approach, we have obtained a fixed mapping of the CC on the computation nodes with BLIND (design optimization strategy from [Izosimov et al., 2005; Pop et al., 2009]). SBS has produced an end-to-end delay of 384 ms. This delay is larger than the deadline. The CS approach, proposed in this paper, reduces this delay to 346 ms, given that all inter-processor messages are frozen. This delay is also unschedulable. If we relax this transparency requirement and select 50% of the inter-processor messages as frozen, we will further reduce the delay to 274 ms that will meet the deadline.

For the evaluation of our mapping algorithm, we have compared BLIND to AWARE (our proposed transparency aware mapping strategy). The solution obtained with BLIND is schedulable only with 50% frozen messages and no frozen processes. However, the application optimized with AWARE, is easily schedulable with 85% frozen messages. Moreover, we can additionally introduce 20% frozen processes without violating the deadlines.

9. CONCLUSIONS

In this paper, we have presented an approach to synthesizing efficient fault-tolerant schedules for distributed real-time embedded systems in the presence of multiple transient faults. The approach supports fine-grained customized transparency. Transparency has the advantages of improved debuggability and less memory needed to store the fault-tolerant schedules. The proposed scheduling algorithm has the ability to handle fine-grained, process and message level, transparency requirements. This provides an opportunity for the designer to handle performance versus transparency and memory size trade-offs.

We have also proposed a mapping optimization strategy for applications with transparency requirements. Since the conditional scheduling algorithm is computation-intensive and cannot be used inside an optimization loop, we have proposed a fast estimation heuristic that is able to accurately evaluate a given mapping decision. The proposed mapping algorithm, based on the estimation heuristic, is able to produce effective design solutions for a given transparency setup.

Considering the fault-tolerance and transparency requirements during design optimization, we are able to deliver efficient solutions with increased debuggability under limited amount of available resources.

10. REFERENCES

AHN, K.D., KIM, J., and HONG, S.J. 1997. Fault-Tolerant Real-Time Scheduling Using Passive Replicas. In *Proc. Pacific Rim Intl. Symp. on Fault-Tolerant Systems*, 98-103.

AL-OMARI, R., SOMANI, A.K., and MANIMARAN, G. 2001. A New Fault-Tolerant Technique for Improving Schedulability in Multiprocessor Real-Time Systems. In *Proc. 15th Intl. Parallel and Distributed Processing Symp.*, 23-27.

ALSTROM, K., and TORIN, J. 2001. Future Architecture for Flight Control Systems. In *Proc. 20th Conf. on Digital Avionics Systems*, 1B5/1-1B5/10.

AYAV, T., FRADET, P., and GIRAULT, A. 2008. Implementing Fault-Tolerance in Real-Time Programs by Automatic Program Transformations. *ACM Trans. on Embedded Computing Systems 7(4)*, 1-43.

BALAKIRSKY, V.B., and VINCK, A.J.H. 2006. Coding Schemes for Data Transmission over Bus Systems. In *Proc. IEEE Intl. Symp. on Information Theory*, 1778-1782.

BENSO, A., DI CARLO, S., DI NATALE, G., and PRINETTO, P. 2003. A Watchdog Processor to Detect Data and Control Flow Errors. In *Proc. 9th IEEE On-Line Testing Symposium*, 144-148.

BERTOSSI, A., and MANCINI, L. 1994. Scheduling Algorithms for Fault-Tolerance in Hard-Real Time Systems. *Real Time Systems 7(3)*, 229–256.

BOURRET, P., FERNANDEZ, A., and SEGUIN, C. 2004. Statistical Criteria to Rationalize the Choice of Run-Time Observation Points in Embedded Software. In *Proc. 1st Intl. Workshop on Testability Assessment*, 41-49.

BURNS, A., DAVIS, R., and PUNNEKKAT, S. 1996. Feasibility Analysis of Fault-Tolerant Real-Time Task Sets. In *Proc. Euromicro Workshop on Real-Time Systems*, 29–33.

CHEVOCHOT, P., and PUAUT, I. 1999. Scheduling Fault-Tolerant Distributed Hard Real-Time Tasks Independently of the Replication Strategies. In *Proc. 6th Intl. Conf. on Real-Time Computing Systems and Applications*, 356-363.

CHANDRA, T.D., and TOUEG, S. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *J. of the ACM 43(2)*, 225-267.

CLAESSON, V., POLEDNA, S., and SODERBERG, J. 1998. The XBW Model for Dependable Real-Time Systems. In *Proc. Intl. Conf. on Parallel and Distributed Systems*, 130-138.

CONNER, J., XIE, Y., KANDEMIR, M., LINK, G., and DICK, R. 2005. FD-HGAC: A Hybrid Heuristic/Genetic Algorithm Hardware/Software Co-synthesis Framework with Fault Detection. In *Proc. Asia and South Pacific Design Automation Conf. (ASP-DAC)*, 709-712.

CONSTANTINESCU, C. 2003. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro 23(4)*, 14-19.

ELES, P., DOBOLI, A., POP, P., and PENG, Z. 2000. Scheduling with Bus Access Optimization for Distributed Embedded Systems. *IEEE Trans. on Very Large Scale Integrated (VLSI) Systems 8(5)*, 472-491.

EMANI, K.C., KAM, K., and ZAWODNIOK, M. 2007. Improvement of CAN BUS Performance by Using Error-Correction Codes. In *Proc. IEEE Region 5 Technical Conf.*, 205-210, *AR*.

Girault, A., Kalla, H., Sighireanu, M., and SOREL, Y. 2003. An Algorithm for Automatically Obtaining Distributed and Fault-Tolerant Static Schedules, In *Proc. Intl. Conf. on Dependable Systems and Networks (DSN)*, 159-168.

HAN, C. C., SHIN, K.G., and WU, J. 2003. A Fault-Tolerant Scheduling Algorithm for Real-Time Periodic Tasks with Possible Software Faults. *IEEE Trans. on Computers 52(3)*, 362–372.

HAN, J.-J., and LI, Q.-H. 2005. Dynamic Power-Aware Scheduling Algorithms for Real-Time Task Sets with Fault-Tolerance in Parallel and Distributed Computing Environment. In *Proc. 19th IEEE Intl. Parallel and Distributed Processing Symp.*, 6-16.

HARELAND, S., MAIZ, J., ALAVI, M., MISTRY, K., WALSTA, S., and DAI, C.H.. 2001. Impact of CMOS Process Scaling and SOI on the Soft Error Rates of Logic Processes. In *Proc. Symp. on VLSI Technology*,73-74.

HEINE, P., TURUNEN, J., LEHTONEN, M., and OIKARINEN, A. 2005. Measured Faults during Lightning Storms. In *Proc. IEEE PowerTech'2005, Paper 72*, 5pp.
IZOSIMOV, V., POP, P., ELES, P., and PENG, Z. 2005. Design Optimization of Time- and Cost-Constrained Fault-Tolerant Distributed Embedded Systems. In *Proc. Design Automation and Test in Europe Conf. (DATE)*, 864-869.
Izosimov, V., Pop, P., Eles, P., and PENG, Z. 2006a. Synthesis of Fault-Tolerant Schedules with Transparency/ Performance Trade-offs for Distributed Embedded Systems. In *Proc. Design Automation and Test in Europe Conf. (DATE)*, 706-711.
IZOSIMOV, V., POP, P., ELES, P., and PENG, Z. 2006b. Mapping of Fault-Tolerant Applications with Transparency on Distributed Embedded Systems. In *Proc. 9th Euromicro Conf. on Digital System Design (DSD)*, 313-320.
IZOSIMOV, V. 2009. Scheduling and Optimization of Fault-Tolerant Distributed Embedded Systems, In PhD Thesis No. 1290, Dept. of Computer and Information Science, Linköping University, Linköping, Sweden.
Permanent link: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-51727
JUNIOR, D.B., VARGAS, F., SANTOS, M.B., TEIXEIRA, I.C., and TEIXEIRA, J.P. 2004. Modeling and Simulation of Time Domain Faults in Digital Systems. In *Proc. 10th IEEE Intl. On-Line Testing Symp.*, 5-10.
KANDASAMY, N., HAYES, J.P., and MURRAY, B.T. 2003a. Transparent Recovery from Intermittent Faults in Time-Triggered Distributed Systems. *IEEE Trans. on Computers 52(2)*, 113–125.
KANDASAMY, N., HAYES, J.P., and MURRAY, B.T. 2003b. Dependable Communication Synthesis for Distributed Embedded Systems. In *Proc. Computer, Safety, Reliability and Security Conf.*, 275–288.
KOPETZ, H., KANTZ, H., GRUNSTEIDL, G., PUSCHNER, P., and REISINGER, J. 1990. Tolerating Transient Faults in MARS. In *Proc. 20th Intl. Symp. on Fault-Tolerant Computing*, 466-473.
KOPETZ, H., and BAUER, G. 2003. The Time-Triggered Architecture. *Proc. of the IEEE 91(1)*, 112–126.
KOPETZ, H., OBERMAISSER, R., PETI, P., and SURI, N. 2004. From a Federated to an Integrated Architecture for Dependable Embedded Real-Time Systems. In *Technical Report 22, Technische Universität Wien, Vienna, Austria*.
KOREN, I., and KRISHNA, C.M. 2007. Fault-Tolerant Systems. *Morgan Kaufmann Publishers, Elsevier, San Francisco, CA.*
KRISHNA, C.M., and SINGH, A.D. 1993. Reliability of Checkpointed Real-Time Systems Using Time Redundancy. *IEEE Trans. on Reliability 42(3)*, 427-435.
LATIF-SHABGAHI, G.. BASS, J.M., and BENNETT, S. 2004. A Taxonomy for Software Voting Algorithms Used in Safety-Critical Systems. *IEEE Trans. on Reliability 53(3)*, 319-328.
LEE, H., SHIN, H., and MIN, S.-L. 1999. Worst Case Timing Requirement of Real-Time Tasks with Time Redundancy. In *Proc. 6th Intl. Conf. on Real-Time Computing Systems and Applications*, 410-414.
LIBERATO, F., MELHEM, R., and MOSSE, D. 2000. Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard Real-Time Systems. *IEEE Trans. on Computers 49(9)*, 906–914.
MAHESHWARI, A., BURLESON, W., and TESSIER, R. 2004. Trading Off Transient Fault Tolerance and Power Consumption in Deep Submicron (DSM) VLSI Circuits. *IEEE Trans. on Very Large Scale Integrated (VLSI) Systems 12(3)*, 299-311.
MAY, T.C., and WOODS, M.H. 1978. A New Physical Mechanism for Soft Error in Dynamic Memories. In *Proc. 16th Intl. Reliability Physics Symp.*, 33-40.
MELHEM, R., MOSSE, D., and ELNOZAHY, E. 2004. The Interplay of Power Management and Fault Recovery in Real-Time Systems. *IEEE Trans. on Computers 53(2)*, 217-231.
METRA, C., FAVALLI, M., and RICCO, B. 1998. On-line Detection of Logic Errors due to Crosstalk, Delay, and Transient Faults. In *Proc. Intl. Test Conf. (ITC)*, 524-533.
NICOLESCU, B., SAVARIA, Y., and VELAZCO, R. 2004. Software Detection Mechanisms Providing Full Coverage against Single Bit-Flip Faults. *IEEE Trans. on Nuclear Science*, 51(6), 3510-3518.
OH, N., SHIRVANI, P.P., and MCCLUSKEY, E.J. 2002. Control-Flow Checking by Software Signatures. *IEEE Trans. on Reliability 51(2)*, 111-122.
OH, N., SHIRVANI, P.P., and MCCLUSKEY, E.J. 2002. Error Detection by Duplicated Instructions in Super-Scalar Processors. *IEEE Trans. on Reliability 51(1)*, 63-75.
ORAILOGLU, A., and KARRI, R. 1994. Coactive Scheduling and Checkpoint Determination during High Level Synthesis of Self-Recovering Microarchitectures. *IEEE Trans. on Very Large Scale Integrated (VLSI) Systems 2(3)*, 304-311.
PARMER, G., and WEST, R. 2007. Mutable Protection Domains: Towards a Component-Based System for Dependable and Predictable Computing. In *Proc. 28th IEEE Intl .Real-Time Systems Symposium (RTSS)*, 365-378.
PINELLO, C., CARLONI, L.P., and SANGIOVANNI-VINCENTELLI, A.L. 2004. Fault-Tolerant Deployment of Embedded Software for Cost-Sensitive Real-Time Feedback-Control Applications. In *Proc. Design, Automation and Test in Europe Conf. (DATE)*, 1164–1169.
PINELLO, C., CARLONI, L.P., and SANGIOVANNI-VINCENTELLI, A.L. 2008. Fault-Tolerant Distributed Deployment of Embedded Control Software. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 27(5)*, 906-919.
PIRIOU, E., JEGO, C., ADDE, P., LE BIDAN, R., and JEZEQUEL, M. 2006. Efficient Architecture for Reed Solomon Block Turbo Code. In *Proc. IEEE Intl. Symp. on Circuits and Systems (ISCAS)*, 4 pp.

POLEDNA, S. 1995. Fault Tolerant Real-Time Systems – The Problem of Replica Determinism. *Springer, Dordrecht, The Netherlands.*
POP, P., ELES, P., and PENG, Z. 2004. Analysis and Synthesis of Distributed Real-Time Embedded Systems. *Kluwer Academic Publishers, Dordrecht, The Netherlands.*
POP, P., ELES, P., and PENG, Z. 2005. Schedulability-Driven Frame Packing for Multi-Cluster Distributed Embedded Systems. *ACM Trans. on Embedded Computing Systems 4(1)*, 112-140.
POP, P., IZOSIMOV, V., ELES, P., and PENG, Z. 2009. Design Optimization of Time- and Cost-Constrained Fault-Tolerant Embedded Systems with Checkpointing and Replication. *IEEE Trans. on Very Large Scale Integrated (VLSI) Systems 17(3)*, 389-402.
POP, P., POULSEN, K.H., IZOSIMOV, V., and ELES, P. 2007. Scheduling and Voltage Scaling for Energy/ Reliability Trade-offs in Fault-Tolerant Time-Triggered Embedded Systems. In *Proc. 5th IEEE/ACM Intl. Conf. on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 233-238.
PUNNEKKAT, S., and BURNS, A. 1997. Analysis of Checkpointing for Schedulability of Real-Time Systems. In *Proc. 4th Intl. Workshop on Real-Time Computing Systems and Applications*, 198-205.
PUSCHNER, P., and BURNS, A. 2000. Guest Editorial: A Review of Worst-Case Execution-Time Analysis. *Real-Time Systems 18(2-3)*, 115-128.
REEVS, C.R. 1993. Modern Heuristic Techniques for Combinatorial Problems. *Blackwell Scientific Publications, Oxford, U.K.*
ROSSI, D., OMANA, M., TOMA, F., and METRA, C. 2005. Multiple Transient Faults in Logic: An Issue for Next Generation ICs? In *Proc. 20th IEEE Intl. Symp. on Defect and Fault Tolerance in VLSI Systems*, 352-360.
SAVOR, T., and SEVIORA, R.E. 1997. An Approach to Automatic Detection of Software Failures in Real-Time Systems. In *Proc. 3rd IEEE Real-Time Technology and Applications Symp*., 136-146.
SCIUTO, D., SILVANO, C., and STEFANELLI, R. 1998. Systematic AUED Codes for Self-Checking Architectures. In *Proc. IEEE Intl. Symp. on Defect and Fault Tolerance in VLSI Systems*, 183-191.
SHIVAKUMAR, P., KISTLER, M., KECKLER, S.W., BURGER, D., and ALVISI, L. 2002. Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic. In *Proc. Intl. Conf. on Dependable Systems and Networks (DSN)*, 389-398.
SILVA, V.F., FERREIRA, J., and FONSECA, J.A. 2007. Master Replication and Bus Error Detection in FTT-CAN with Multiple Buses. In *Proc. IEEE Conf. on Emerging Technologies & Factory Automation (ETFA)*, 1107-1114.
SRINIVASAN, S., and JHA, N.K. 1995. Hardware-Software Co-Synthesis of Fault-Tolerant Real-Time Distributed Embedded Systems. In *Proc. of Europe Design Automation Conf.*, 334-339.
SHYE, A., MOSELEY, T., REDDI, V.J., BLOMSTEDT, J., and CONNORS, D.A. 2007. Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance. In *Proc. Intl. Conf. on Dependable Systems and Networks (DSN)*, 297-306.
STRAUSS, B., MORGAN, M.G., APT, J., and STANCIL, D.D. 2006. Unsafe at Any Airspeed? *IEEE Spectrum 43(3)*, 44-49.
TRIPAKIS, S. 2005. Two-phase Distributed Observation Problems. In *Proc. 5th Intl. Conf. on Application of Concurrency to System Design*, 98-105.
TSAI, T. 1998. Fault Tolerance via N-modular Software Redundancy. In *Proc. 28th Annual Intl. Symp. on Fault-Tolerant Computing*, 201-206.
ULLMAN, D. 1975. NP-Complete Scheduling Problems. *Computer Systems Science 10*, 384–393.
VELAZCO, R., FOUILLAT, P., and REIS, R. *(Eds.)*. 2007. Radiation Effects on Embedded Systems. *Springer, Dordrecht, The Netherlands.*
VRANKEN, H.P.E., STEVENS, M.P.J., and SEGERS, M.T.M. 1997. Design-for-Debug in Hardware/Software Co-Design. In *Proc. 5th Intl. Workshop on Hardware/Software Codesign*, 35-39.
WANG, J.B. 2003. Reduction in Conducted EMI Noises of a Switching Power Supply after Thermal Management Design. *IEE Proc. – €€Electric Power Applications 150(3)*, 301-310.
WEI, H., STAN, M.R., SKADRON, K., SANKARANARAYANAN, K., GHOSH, S., and VELUSAMY, S. 2004. Compact Thermal Modeling for Temperature-aware Design. In *Proc. Design Automation Conference (DAC)*, 878-883.
WEI, T., MISHRA, P., WU, K., and LIANG, H. 2006. Online Task-Scheduling for Fault-Tolerant Low-Energy Real-Time Systems. In *Proc. IEEE/ACM Intl. Conf. on Computer-Aided Design (ICCAD)*, 522-527.
XIE, Y., LI, L., KANDEMIR, M., VIJAYKRISHNAN, N., and IRWIN, M.J. 2004. Reliability-aware Co-synthesis for Embedded Systems. In *Proc. 15th IEEE Intl. Conf. on Application-Specific Systems, Architectures and Processors (ASAP)*, 41-50.
XIE, Y. , LI, L., KANDEMIR, M., VIJAYKRISHNAN, N., and IRWIN, M.J. 2007. Reliability-Aware Co-synthesis for Embedded Systems. *The J. of VLSI Signal Processing 49(1)*, 87-99.
XU, J., and RANDELL, B. 1996. Roll-Forward Error Recovery in Embedded Real-Time Systems. In *Proc. Intl. Conf. on Parallel and Distributed Systems*, 414-421.
ZHANG, Y., and CHAKRABARTY, K. 2006. 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.
ZHU, D., MELHEM, R., and MOSSÉ, D. 2005. Energy Efficient Configuration for QoS in Reliable Parallel Servers. In *Lecture Notes in Computer Science 3463*, 122-139.

11. APPENDIX I: FTPG GENERATION

In Fig. 15 we have outlined the BuildFTPG algorithm that traces processes in the merged graph G with transparency requirements T in the presence of maximum k faults and

BuildFTPG($\mathcal{G}$, $\mathcal{T}$, k)
1 $G = \emptyset$
2 P_i = RootNode($\mathcal{G}$); Insert(P_i^1, G); $faults(P_i^1) = k$ -- **insert the root node**
3 **for** $f = k - 1$ **downto** 0 **do** -- **insert re-executions of the root node**
4 Insert(P_i^{k-f+1}, G); Connect(P_i^{k-f}, P_i^{k-f+1}); $faults(P_i^{k-f+1}) = f$
5 **end for**
6 $\mathcal{L} = \emptyset$ -- **add successors of the root node to the process list**
7 **for** $\forall$Succ(P_i) $\in \mathcal{G}$ **do** $\mathcal{L} = \mathcal{L} \cup$ Succ(P_i)
8 **while** $\mathcal{L} \neq \emptyset$ **do** -- **trace all processes in the merged graph** $\mathcal{G}$
9 P_i = ExtractProcess($\mathcal{L}$)
10 $\mathcal{VC}$ = GetValidPredCombinations(P_i, G)
11 **for** $\forall m_j \in$ InputMessages(P_i) **if** $\mathcal{T}(m_j) \equiv Frozen$ **do** -- **transform frozen messages**
12 Insert(m_j^S, G) -- **insert "message" synchronization node**
13 **for** $\forall vc_n \in \mathcal{VC}$ **do**
14 Connect($\forall P_x^m \{m_j\} \in vc_n$, m_j^S)
15 **end for**
16 UpdateValidPredCombinations($\mathcal{VC}$, G)
17 **end for**
18 **if** $\mathcal{T}(P_i) \equiv Frozen$ **then** -- **if process P_i is frozen, then insert corresponding synchronization node**
19 Insert(P_i^S, G) -- **insert "process" synchronization node**
20 **for** $\forall vc_n \in \mathcal{VC}$ **do**
21 Connect($\forall P_x^m \in vc_n$, P_i^S); Connect($\forall m_x^S \in vc_n$, P_i^S)
22 **end for**
23 Insert(P_i^1, G); Connect(P_i^1, $\forall P_x^m \in vc_n$); $faults(P_i^1) = k$ -- **insert first copy of P_i**
24 **for** $f = k - 1$ **downto** 0 **do** -- **insert re-executions**
25 Insert(P_i^{k-f+1}, G); Connect(P_i^{k-f}, P_i^{k-f+1}); $faults(P_i^{k-f+1}) = f$
26 **end for**
27 **else** -- **if process P_i is regular**
28 $h = 1$
29 **for** $\forall vc_n \in \mathcal{VC}$ **do** -- **insert copies of process P_i**
30 Insert(P_i^h, G); Connect($\forall P_x^m \in vc_n$, P_i^h); Connect($\forall m_x^S \in vc_n$, P_i^h)
31 **if** $\exists m_x^S \in vc_n$ **then** $faults(P_i^h) = k$
32 **else** $faults(P_i^h) = k - \sum_{\forall P_x^m \in vc_n} (k - faults(P_x^m))$
33 **end if**
34 $f = faults(P_i^h) - 1$; $h = h + 1$
35 **while** $f \geq 0$ **do** -- **insert re-executions**
36 Insert(P_i^h, G); Connect(P_i^h, P_i^{h-1}); $faults(P_i^h) = f$; $f = f - 1$; $h = h + 1$
37 **end while**
38 **end for**
39 **end if**
40 **for** $\forall$Succ(P_i) $\in \mathcal{G}$ **do** -- **add successors of process P_i to the process list**
41 **if** $\forall$Succ(P_i) $\notin \mathcal{L}$ **then** $\mathcal{L} = \mathcal{L} \cup$ Succ(P_i)
42 **end for**
43 **end while**
44 **return** G
end BuildFTPG

Fig. 15. Generation of FTPG.

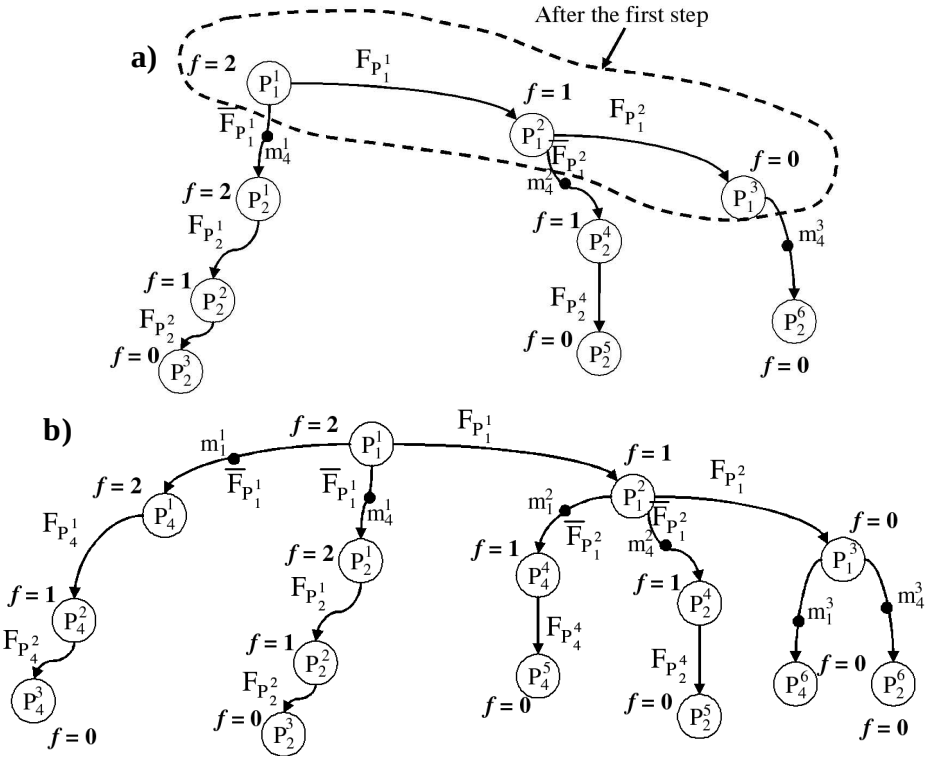


Fig. 16. FTPG Generation Steps (1).

generates the corresponding FTPG *G*. In the first step, BuildFTPG copies the root process into the FTPG (line 2). Then, re-executions of the root process are inserted, connected through "faulty" conditional edges with the "true" condition value (lines 3–5). Copies of the root process (including its re-executions) are assigned with f, $f-1$, $f-2$,..., 0 *possible* faults, respectively, where $f = k$ for the root process. These fault values will be used in the later construction steps. In Fig. 16a, we show the intermediate state resulted after this first step during the generation of the FTPG depicted in Fig. 9b. After the first step, copies P_1^1, P_1^2 and P_1^3 are inserted (where $k = 2$), connected with the conditional edges e_{11}^{12} and e_{11}^{23}, between copies P_1^1 and P_1^2, and between copies P_1^2 and P_1^3, respectively. Copies P_1^1, P_1^2 and P_1^3 are assigned with $f = 2$, $f = 1$ and $f = 0$ possible faults, as shown in the figure.

In the next step, BuildFTPG places successors of the root process into the process list *L* (line 7). For generation of the FTPG, the order of processes in the process list *L* is not important and BuildFTPG extracts the first available process P_i (line 9). By an "available"

process, we denote a process P_i with all its predecessors already incorporated into the FTPG *G*.

For each process P_i, extracted from the process list *L*, BuildFTPG prepares a set *VC* of *valid* combinations of copies of the predecessor processes (line 10). A combination is valid (1) if the copies of predecessor processes in each combination $vc_n \in VC$ correspond to a non-conflicting set of condition values, (2) if all copies of the predecessors together, do not accumulate more than *k* faults, and (3) if the combination contains at most one copy of each predecessor.

Let us extract process P_2 from the process list *L* and incorporate this process into the FTPG *G*. In the application graph *G*, process P_2 has only one predecessor P_1. Initially, the set of combinations of copies of predecessors for process P_2 will contain seven elements: $\{ P_1^1 \}$, $\{ P_1^2 \}$, $\{ P_1^3 \}$, $\{ P_1^1, P_1^2 \}$, $\{ P_1^1, P_1^3 \}$, $\{ P_1^2, P_1^3 \}$ and $\{ P_1^1, P_1^2, P_1^3 \}$.

According to the first rule, none of the elements in this set corresponds to a conflicting set of conditional values. For example, for $\{ P_1^1, P_1^2, P_1^3 \}$, P_1^1 is activated upon the condition *true* as the root node of the graph; P_1^2 under condition $F_{P_1^1}$; and P_1^3 under joint condition $F_{P_1^1} \wedge F_{P_1^2}$. Condition *true* is not in conflict with any of the conditions. Conditions $F_{P_1^1}$ and $F_{P_1^1} \wedge F_{P_1^2}$ are not in conflict since $F_{P_1^1} \wedge F_{P_1^2}$ includes $F_{P_1^1}$.[9] If, however, we apply the second rule, $\{ P_1^2, P_1^3 \}$ and $\{ P_1^1, P_1^2, P_1^3 \}$ are not valid since they would accumulate more than $k = 2$ faults, i.e., 3 faults each. Finally, only three elements $\{ P_1^1 \}$, $\{ P_1^2 \}$ and $\{ P_1^3 \}$ satisfy the last rule. Thus, in Fig. 16a, the set of valid predecessors *VC* for process P_2 will contain three elements with copies of process P_1: $\{ P_1^1 \}$, $\{ P_1^2 \}$, and $\{ P_1^3 \}$.

In case of any frozen input message to P_2, we would need to further modify this set *VC*, in order to capture transparency properties. However, since all input messages of process P_2 are regular, the set of combinations should not be modified, i.e., we skip lines 12-16 in the BuildFTPG and go directly to the process incorporation step.

---

[9] An example of conflicting conditions, for example, would be $F_{P_1^1} \wedge F_{P_1^2}$ and $F_{P_1^1} \wedge \overline{F}_{P_1^2}$ that contain mutually exclusive condition values $F_{P_1^2}$ and $\overline{F}_{P_1^2}$.

For regular processes, such as P_2, the FTPG generation proceeds according to lines 28– 38 in Fig. 15. For each combination $vc_n \in VC$, BuildFTPG inserts a corresponding copy of process P_i, connects it to the rest of the graph (line 30) with conditional and unconditional edges that carry copies of input messages to process P_i, and assigns the number of possible faults (lines 31–33). If the combination vc_n contains a "message" synchronization node, the number of possible faults f for the inserted copy will be set to the maximum k faults (line 31). Otherwise, f is derived from the number of possible faults in all of the predecessors' copies $P_x^m \in vc_n$ as $f(P_i^h) = k - \sum(k - f(P_x^m))$ (line 32). In this formula, we calculate how many faults have already happened before invocation of P_i^h, and then derive the number of faults that *can* still happen (out of the maximum k faults). Once the number of possible faults f is obtained, BuildFTPG inserts f re-execution copies that will be invoked to tolerate these faults (lines 34–37). Each re-execution copy P_i^h is connected to the preceding copy P_i^{h-1} with a "faulty" conditional edge e_{ii}^{h-1h}. The number of possible faults for P_i^h is, consequently, reduced by 1, i.e., $f(P_i^h) = f(P_i^{h-1}) - 1$.

In Fig. 16a, after P_1^1, with $f = 2$, copies P_2^1, P_2^2 and P_2^3 are inserted, connected with the conditional edges e_{12}^{11}, e_{22}^{12} and e_{22}^{23}, that will carry copies m_4^1, m_4^2 and m_4^3 of message m_4. After P_1^2, with $f = 1$, copies P_2^4 and P_2^5 are inserted, connected with the conditional edges e_{12}^{24} and e_{22}^{45}. After P_1^3, with no more faults possible ($f = 0$), a copy P_2^6 is introduced, connected to P_1^3 with the unconditional edge e_{12}^{36}. This edge will be always taken after P_1^3. The number of possible faults for P_2^1 is $f = 2$. For re-execution copies P_2^2 and P_2^3, $f = 1$ and $f = 0$, respectively. The number of possible faults for P_2^4 is $f = 1$. Hence, $f = 0$ for the corresponding re-execution copy P_2^5. Finally, no more faults are possible for P_2^6, i.e., $f = 0$.

In Fig. 16b, process P_4 is also incorporated into the FTPG G, with its copies connected to the copies of P_1. Edges e_{14}^{11}, e_{14}^{24} and e_{14}^{36}, which connect copies of P_1 (P_1^1, P_1^2, and P_1^3) and copies of P_4 (P_4^1, P_4^4, and P_4^6), will carry copies m_1^1, m_1^2 and m_1^3 of message m_1.

When process P_i has been incorporated into the FTPG G, its available successors are placed into the process list L (lines 40–42). For example, after P_2 and P_4 have been incorporated, process P_3 is placed into the process list L. BuildFTPG continues until all processes and messages in the merged graph G are incorporated into the FTPG G, i.e., until the list L is empty (line 8).

After incorporating processes P_1, P_2 and P_4, the process list L will contain only process P_3. Contrary to P_2 and P_4, the input of process P_3 includes two frozen messages m_2 and m_3. Moreover, process P_3 is itself frozen. Thus, the procedure of incorporating P_3 into the FTPG G will proceed according to lines 19-26 in Fig. 15. In the application graph G, process P_3 has three predecessors P_1, P_2, and P_4. Thus, its set of valid combinations VC of copies of the predecessor processes will be as: $\{P_1^1, P_2^1, P_4^1\}$, $\{P_1^1, P_2^1, P_4^2\}$, $\{P_1^1, P_2^1, P_4^3\}$, $\{P_1^1, P_2^2, P_4^1\}$, $\{P_1^1, P_2^2, P_4^2\}$, $\{P_1^1, P_2^3, P_4^1\}$, $\{P_1^2, P_2^4, P_4^4\}$, $\{P_1^2, P_2^4, P_4^5\}$, $\{P_1^2, P_2^5, P_4^4\}$ and $\{P_1^3, P_2^6, P_4^6\}$.

If any of the input messages of process P_i is frozen (line 11), the corresponding synchronization nodes are inserted and connected to the rest of the nodes in G (lines 12–15). In this case, the set of valid predecessors VC is updated to include the synchronization nodes (line 17). Since input messages m_2 and m_3 are frozen, two synchronization nodes m_2^S and m_3^S are inserted, as illustrated in Fig. 17. m_2^S and m_3^S are connected to the copies of the predecessor processes with the following edges: $e_{12}^{1S_m}$,

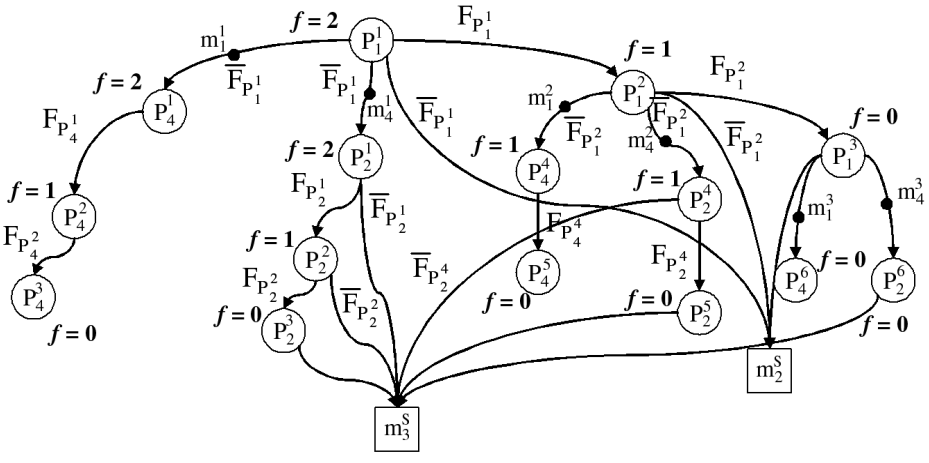


Fig. 17. FTPG Generation Steps (2).

$e_{12}^{2S_m}$, and $e_{12}^{3S_m}$ (for m_2^S), and $e_{23}^{1S_m}$, $e_{23}^{2S_m}$, $e_{23}^{3S_m}$, $e_{23}^{4S_m}$, $e_{23}^{5S_m}$, and $e_{23}^{6S_m}$ (for m_3^S). The set of valid predecessors *VC* is updated to include the synchronization nodes: { m_2^S, m_3^S, P_4^1 }, { m_2^S, m_3^S, P_4^2 }, { m_2^S, m_3^S, P_4^3 }, { m_2^S, m_3^S, P_4^4 }, { m_2^S, m_3^S, P_4^5 }, and { m_2^S, m_3^S, P_4^6 }. Note that the number of combinations has been reduced due to the introduction of the synchronization nodes.

Since process P_3 is frozen, we first insert synchronization node P_3^S (line 19), as illustrated in Fig. 9b, which is connected to the copies of the predecessor processes and the other synchronization nodes with the edges e_{43}^{1S}, e_{43}^{2S}, e_{43}^{3S}, e_{43}^{4S}, e_{43}^{5S}, e_{43}^{6S}, $e_{23}^{S_mS}$ and $e_{33}^{S_mS}$ (lines 20–22). After that, the first copy P_3^1 of process P_3 is inserted, assigned with $f = 2$ possible faults (line 23). P_3^1 is connected to the synchronization node P_3^S with edge e_{33}^{S1}. Finally, re-execution copies P_3^2 and P_3^3 with $f = 1$ and $f = 0$ possible faults, respectively, are introduced, and connected with two "faulty" conditional edges e_{33}^{12} and e_{33}^{23} (lines 24– 26), which leads to the complete FTPG *G* depicted in Fig. 9b. The algorithm will now terminate since process P_3 is the last process in the graph *G*.